

# Partitioning the Threads of a Mobile System

Jérôme Feret  
 École Normale Supérieure  
 Harvard Medical School  
 jerome\_feret@hms.harvard.edu

February 4, 2008

## Abstract

In this paper, we show how thread partitioning helps in proving properties of mobile systems. Thread partitioning consists in gathering the threads of a mobile system into several classes. The partitioning criterion is left as a parameter of both the mobility model and the properties we are interested in. Then, we design a polynomial time abstract interpretation-based static analysis that counts the number of threads inside each partition class.

## 1 Introduction

A mobile system is a pool of threads that interact with each other. These interactions dynamically change the system by controlling both the creation and the destruction of links between threads (by modifying the accesses to channels and/or modifying the spatial configuration). These interactions also control the creation of threads. The size of a mobile system may be unbounded. A mobile system may describe telecommunication networks, reconfigurable systems, *client-server* applications, cryptographic protocols, or biological systems. Several models exist according to the application field and the granularity of the observation level.

We use abstract interpretation [11, 13] to derive abstract semantics, which are sound, decidable, but approximate. We use partitioning [14, 6] to separate the threads according to dynamical information. The partitioning criterion depends on both the model and the properties of interest. In models based on channeled communications (as in the  $\pi$ -calculus [29]), we can partition the threads according to the name of the channel they operate on. In models with explicit locations (as in *ambients* [9]), we can partition the threads according to their location in the system. When there are both channeled communications and locations (as in D- $\pi$  [36] or in BIO-*ambients* [35]), we partition the threads according to both the channel they operate on and their location. In more complex cases, the partitioning criterion may be given manually. For instance, in

the *spi*-calculus, channels are not relevant, so we partition the threads according to the principals that share a session [22, p:269] thanks to some end-user’s annotations. Nevertheless, we believe that a better understanding of the problem should allow the automatic inference of these annotations.

Our analysis then counts automatically the number of threads inside each partition class. To get an accurate analysis, we have to relate, for each computation step, the partition classes of the threads that interact and the partition classes of the threads that are created. When analyzing mobile *ambients* [33], these relations are given by the model. This is not the case in less structured models, where a non uniform (i.e. that distinguishes recursive instances) analysis [18, 20, 21, 22] of the dynamic linkage between threads is required. To make contents analysis and non uniform analysis collaborate, we locally partition computation steps [28] according to some assumptions about the partition classes of the threads that interact. Then, we use a coalesced product between both analyses, so that if one detects that some assumptions are contradictory, the other ignores the corresponding interaction.

We apply our framework to prove automatically the absence of race conditions in a shared-memory with dynamic allocation written in the  $\pi$ -calculus. We also analyze precisely the relation between the contents of an *ambient* and its location in the network. In the author’s PhD. Thesis [22], we prove an authentication property [3] in a cryptographic protocol [38] in the *spi*-calculus [1].

**Outline.** We discuss related works in Sect. 2. We detail the contribution of this paper in Sect. 3. We give some examples in Sect. 4. We give in Sect. 5 a non-standard semantics for the  $\pi$ -calculus. We define both thread and step partitioning in Sect. 6. We derive a generic abstraction in Sect. 7. We give an environment analysis in Sect. 8 and a contents analysis in Sect. 9.

## 2 Related works

In this section, we discuss some related works.

### 2.1 Control flow analyses

Our analysis requires an accurate description of the potential interactions between the agents. Many type systems [25] and control flow analyses [5, 4] propose a uniform description of these interactions in which recursive instances cannot be distinguished. In [21, 18, 20], we proposed non-uniform control flow analyses, which distinguish between recursive instances of names. All these analyses abstract away the properties about concurrency.

## 2.2 Groups

Groups [8, 7] are used in type system to prevent certain communications. Recursive instances of groups are distinguished. The communication of a name outside the initial scope of its group is forbidden. On the contrary, our analysis computes relationship between the partition classes of interacting threads. So we can analyze systems where a name first exits the scope of the thread that had declared it and then returns inside this scope.

## 2.3 Numerical domains and concurrency

Numeric analyses are widely used to analyze concurrency properties such as mutual exclusion and non-exhaustion of resources. Disjunctive completion-based domains are used in [32] to count globally the components in *ambients* and in [33, 24] to count the components inside each *ambient*. These domains ignore the algebraic structure of numerical properties. Consequently, these analyses are exponential in time. In [21, 19], we use affine equalities to count the threads of  $\pi$ -calculus systems in polynomial time. This analysis counts threads globally, regardless of their linkage. In the present paper, we use information about the dynamic linkage of threads to gather threads in partition classes. Then we count the number of threads inside each partition class. Our approach is model-independent [22]. Besides, we can detect and prove history-dependent and spatial-dependent properties (e.g. see Ex. 4.6).

## 2.4 Behavioral types

Behavioral types can express complex concurrency properties such as the absence of race conditions. But, in [26], some properties involving several names cannot be checked because of the abstraction (e.g. see Ex. 4.2). The type system in [34, 10] can express and check more properties, but the type checking algorithm does not always terminate, whereas our inference algorithm does in polynomial in time. Moreover, our occurrence counting and control flow analyses refine each other thanks to local trace partitioning. In Ex. 4.3, we cannot analyze precisely mutual exclusion without the help of a precise control flow analysis.

## 3 Contribution

In this section, we describe the main contributions of this paper.

This paper is a summary of the framework proposed in [22, Chap. 10]. This framework is generic with respect to the model. In this paper, we focus on systems that are written in the  $\pi$ -calculus. The main contributions of this paper are the following:

1. *thread partitioning*: in this paper, we partition the threads of a mobile system according to some semantics criteria;

2. *local trace partitioning*: then, we provide an extended labeled transition system in which each computation step is annotated with information about the partition classes of the threads that interact; this allows several analyses to share information about the partition classes of the threads that interact;
3. *control flow analysis*: we refine existing analyses [18, 20, 21, 22] so as to take into account the constraints about the partition classes of the threads that interact;
4. *content analysis*: we propose a new analysis to count the number of threads inside each partition class; this analysis is parametric with respect to a numerical domain (we use the same domain as in the occurrence counting analysis [19, 21, 22] that counts the number of threads in the whole system).

## 4 Examples

In this section, we give some examples to motivate our framework.

### 4.1 Our running example

First, we introduce an example that is easy to analyze: we prove that there are never two simultaneous outputs over the same channel in a shared memory written in the  $\pi$ -calculus. We give a manual proof in order to stress the properties that are useful during the analysis. The goal of this example is just to understand how the analysis behaves: we use this example all along the paper.

We use a version of the  $\pi$ -calculus inspired from [29, 37, 2]. Let  $\mathcal{V}$  be an infinite set of variables and  $\mathcal{L}$  be a finite set of labels. Let  $c, x, y \in \mathcal{V}$  be some variables,  $l \in \mathcal{L}$  be a label, and  $\bar{x} \in \mathcal{V}^*$  be a tuple of variables. The agent  $(P \mid Q)$  denotes the parallel composition of two agents  $P$  and  $Q$ . It performs  $P$  and  $Q$  simultaneously. The agent  $(\nu x)P$  binds the variable  $x$  to a fresh channel name in  $P$ . The agent  $\mathbf{0}$  does nothing (it is usually omitted). The agent  $c!^l[\bar{x}].P$  sends a message (i.e. a tuple of channel names) via the channel the name to which the variable  $c$  is bound. The agent  $c?^l[\bar{x}].P$  waits for a message on the channel to which the variable  $c$  is bound, and binds the tuple  $\bar{x}$  of distinct variables to the received names. The agent  $*c?^l[\bar{x}].P$  is a *resource* which replicates itself when receiving messages. Name restriction  $(\nu x)P$  and message reception  $c?^l[\bar{x}].P$  or  $*c?^l[\bar{x}].P$  are the only variable binders. We denote by  $\mathbf{fv}(P)$  the set of the variables that are free in  $P$ . Labels help in locating syntactic components. Moreover, the notation  $\bar{\pi}^l P$  stands for  $(\nu \text{rec}_l)(\text{rec}_l!^l[] \mid * \text{rec}_l?^{l'}[] . (\text{rec}_l!^{l''}[] \mid P))$  where  $l, l'$ , and  $l''$  are fresh labels, and  $\text{rec}_l \notin \mathbf{fv}(P)$ : it denotes an unbounded number of concurrent instances of  $P$ .

**Example 4.1 (a shared memory)** *A shared memory with dynamic allocation of cells may be described in the  $\pi$ -calculus as follows:*

```

(ν alloc)(ν null)
( *alloc?1[address].(ν cell)(ν READ)(ν WRITE)
  ( cell!2[null] | address!3[READ, WRITE]
    | *READ?4[fwd].cell?5[val].(cell!6[val] | fwd!7[val])
    | *WRITE?8[val', ack].cell?9[v].(cell!10[val'] | ack!11[]))
  |  $\bar{*}$ 12(ν add)alloc!13[add].add?14[read, write].
    (  $\bar{*}$ 15(ν return)read!16[return].return?17[x]
      |  $\bar{*}$ 18(ν data)(ν ack)write!19[data, ack].ack?20[]))

```

Whenever a message is sent via the channel name declared by the restriction  $(\nu \text{ alloc})$  (at program point **1**), a memory cell is allocated. Three names are introduced. The name `cell` encodes the contents of the memory cell: the contents of the cell are always output once over the channel named `cell` (the name `null` denotes the initialization value); the names `READ` and `WRITE` encode respectively the capability to read and to overwrite the contents of the cell. The client is given the capability to interact with the cell (at program point **3**). The memory can deal with an unbounded number of read (at program point **4**) and write (at program point **8**) requests. A read request requires a return address to which the contents of the cell are forwarded (please note that we copy the contents of the cell once, so as not to lose them). A write request requires two arguments, the new contents and an acknowledgment address: the cell contents are first removed and then replaced with the new contents, the acknowledgment controls client requests sequentiality. An unbounded number of clients are created (at program point **11**). Each client creates a cell and performs an arbitrary number of read (at program point **15**) and write (at program point **18**) requests.

We want to prove that there is never more than one simultaneous output on any channel  $c$  opened by an instance of the restriction  $(\nu \text{ cell})$ . First, we propose a manual proof to give intuitions about our framework. The analysis in this paper discovers this property automatically. Let us denote by  $\mathcal{M}$  the set of the names introduced by an instance of the restriction  $(\nu \text{ cell})$ , we will prove that at any configuration of the system and any name  $c \in \mathcal{M}$ : there is either no thread, or exactly one output (at program point **2**, **6**, or **10**) on the corresponding channel. For any configuration  $C$  and any name  $c \in \mathcal{M}$ , we define  $y(C, c)$  as 0 whenever the name  $c$  has not been allocated yet, and as 1 otherwise. We denote by  $x_i(C, c)$  the number of threads at program point  $i$  that operate on the channel named  $c$ . Now, we prove by induction over the history of the system that  $x_2(C, c) + x_6(C, c) + x_{10}(C, c) - y(C, c) = 0$ . At the beginning of the system, we have, for any  $c \in \mathcal{M}$ ,  $x_2(C, c) = x_6(C, c) = x_{10}(C, c) = y(C, c) = 0$ , so the property holds. When two threads at program points **1** and **13** interact, a fresh name  $c$  is allocated. Since this name is fresh, we have, before the interaction,  $x_2(C, c) = x_6(C, c) = x_{10}(C, c) = y(C, c) = 0$ ; after the interaction, we have  $x_2(C, c) = y(C, c) = 1$  and  $x_6(C, c) = x_{10}(C, c) = 0$ . This way, the property still holds. We now consider an interaction between a thread  $t_i$  at program point  $i \in \{5; 9\}$  and a thread  $t_j$  at program point  $j \in \{2; 6; 10\}$ ; this interaction launches a thread at program point  $i+1$ . We consider several cases according to the relationships among the channels on which these three threads operate. There

are 5 cases: they may operate on the same channel, on two distinct channels (three cases), or on three distinct channels. We use a control flow analysis to detect which cases are possible: we detect that the only possible case is the case where the three threads operate on the same channel  $c$ . During the transition,  $x_j$  is decremented and  $x_{i+1}$  is incremented ( $x_j$  is not changed when  $j = i + 1$ ), so the property of interest still holds.  $\square$

## 4.2 More complex examples

In this section, we describe more complex examples in order to illustrate some difficulties that can be tackled by our analysis.

**Example 4.2 (related names)** *Our analysis can abstract the usage of several names together. We consider the following system (adapted from [26]) in the  $\pi$ -calculus:*

$$\begin{aligned}
&(\nu b) (*b?^1[c, c', c''] . (\nu l)(\nu m)(\nu r) \\
&\quad (l!^2[] \mid c!^3[l] \mid c'!^4[m] \mid c''!^5[r] \mid \bar{*}^6m?^7[], r!^8[])) \\
&|\bar{*}^9(\nu c)(\nu c')(\nu c'')(b!^{10}[c, c', c''] . \\
&\quad c?^{11}[l].c'?^{12}[m].c''?^{13}[r].\bar{*}^{14}l?^{15}[], m!^{16}[], r?^{17}[], l!^{18}[]))
\end{aligned}$$

The server (at program point **1**) creates several objects. Each object is made of a lock  $l$ , a method  $m$ , and a return address  $r$ . Each session (at program point **15**) consists in locking the method, calling the method, receiving the returned value (which is abstracted away), and then releasing the lock. There is an unbounded number of clients (at program point **9**). Each one creates an object (at program point **10**), receives the lock, the method, and the returned address during three channelled communications, and performs an arbitrary number of sessions (at program point **14**). We partition<sup>1</sup> the threads according to the recursive instance of the resource that has declared the name of the channel on which each thread operates. Our analysis detects and proves that there can never be more than one simultaneous call of the same method. This result is beyond the reach of [26], because the names  $l$ ,  $m$ , and  $r$  are not communicated during a single communication.  $\square$

**Example 4.3 (control flow dependence)** *We now illustrate the importance of the control flow analysis. We describe a doubly-linked list of cells as follows:*

$$\begin{aligned}
&(\nu rec)(\nu l_0)(\nu c_0)(\nu r_0)(\nu SET) \\
&(\text{rec}!^1[l_0, c_0, r_0] \\
&|\text{* rec?}^2[l_n, c_n, r_n].(\nu l_{n+1})(\nu c_{n+1})(\nu r_{n+1}) \\
&\quad (\text{rec}!^3[l_{n+1}, c_{n+1}, r_{n+1}] \mid c_n!^4[] \mid \bar{*}^5SET!^6[l_n, c_n, r_n] \\
&\quad \mid \bar{*}^7l_{n+1}!^8[l_n, c_n, r_n] \mid \bar{*}^9r_n!^{10}[l_{n+1}, c_{n+1}, r_{n+1}]) \\
&|\text{* SET?}^{11}[l, c, r].r?^{12}[l', c', r'].l'?^{13}[l'', c'', r''].c?^{14}[], c''!^{15}[]))
\end{aligned}$$

<sup>1</sup>This partitioning is made possible in the non-standard semantics where each name is tagged with the identifier of the thread that has declared it (see Sect. 5).

Each cell is encoded by three names: the name  $l_n$  encodes a backward pointer to the previous cell, the name  $c_n$  encodes the cell address (the contents are abstracted away), and the name  $r_n$  encodes a forward pointer to the next cell. Each cell is output on the channel named SET (at program point 6). Then, at program point 11, we pick a cell. We collect its address  $c$ , we follow the forward pointer, then we follow the backward pointer, and we collect the address  $c''$  of the reached cell. The control flow analysis [18] detects that the addresses  $c$  and  $c''$  are the same. This information is passed to the occurrence counting domain thanks to the local trace partitioning. Thus, we prove automatically that there is no simultaneous outputs over an instance of a channel named  $c_n$ .

It may look a bit curious to use two variables for the same name. But, these kinds of things are common in automatically generated systems. With a more general point of view, this difficulty is similar to the problem of aliasing in data structures.  $\square$

**Example 4.4 (a 2-semaphore)** Our analysis is not limited to the detection of mutual exclusion. In the following example:

$$\bar{*}^1(\nu a)(a!^2[] \mid a!^3[] \mid *a?^4[].a!^5[]),$$

our analysis detects automatically that there are never more than two simultaneous outputs over an instance of the channel  $a$ . Besides, our analysis detects and proves the number of simultaneous outputs without requiring a bound on the number of copies that have to be distinguished by the analyzer.  $\square$

**Example 4.5 (synchronous communications)** Content analysis can also refine the control flow analysis. In the following system:

$$(\bar{*}^1(\nu a)(\nu b)(\nu c)(a!^2[b].a?^3[u].u!^4[u] \mid a?^5[v].a!^6[c].v!^7[v])),$$

the content analysis detects that, for each instance, the thread at the program point 2 (resp. 6) and the thread at the program point 3 (resp. 5) are in mutual exclusion. The control flow analysis uses this information to prove that the variable  $u$  (resp.  $v$ ) can only be bound to a channel opened by the restriction  $(\nu c)$  (resp.  $(\nu b)$ ).  $\square$

### 4.3 An example in mobile *ambients*

Our last example is written in another process calculus to illustrate that our framework is generic.

In mobile *ambients* [9], a system is described by a hierarchy of named sites  $n^l[P]$ , called *ambients* ( $n$  is a name,  $l$  is a label, and  $P$  is a process). *Ambients* may contain some other *ambients* and some agents  $in^l n.P / out^l .P / open^l n.P$  that provide them the capability to move in the hierarchy of *ambients* or to open some *ambients* (when an *ambient* opens another one, the former *ambient* gets the contents of the later). These interactions are controlled both by *ambient* names (the name of the target *ambient* and the name occurring in the capability

must be the same) and by spatial constraints. These interactions are described by the following reduction rules:  $m^i[in^k n.P \mid Q] \mid n^j[R] \xrightarrow{i,j,k} n^j[m^i[P \mid Q] \mid R]$ ,  $n^j[m^i[out^k n.P \mid Q] \mid R] \xrightarrow{i,j,k} m^i[P \mid Q] \mid n^j[R]$ , and  $open^i n.P \mid n^j[Q] \xrightarrow{i,j} P \mid Q$ . *Ambients* are also fitted with communication primitives: the agent  $(x)^l.P$  waits for a message (that can be either a name or a capability path), whereas the agent  $(y)^l$  sends a message. These communications are not channeled because threads can communicate only when they are in a same *ambient*. Both *ambients* and *ambient* names can be created dynamically. As in the  $\pi$ -calculus, we use guarded replication: the agent  $!(x)^l.P$  duplicates itself when receiving a message and the agent  $!open^l n.P$  duplicates itself when opening another *ambient*.

**Example 4.6 (the contents of an *ambient*)** A client-server protocol may be described in the ambient-calculus as follows:

$$\begin{aligned} &(\nu \text{ make})(\nu \text{ server})(\nu \text{ give\_id})(\nu \text{ instance})(\nu \text{ client}) \\ &(\text{server}^1[!open^2 \text{give\_id}.0 \\ &\quad | !(k)^3.\text{instance}^4[in^5 k.out^6 \text{server}.in^7 \text{client}.0]] \\ &| \text{client}^8[!(x)^9.((\nu p)p^{10}[out^{11} \text{client}.0 | open^{12} \text{instance}.0 \\ &\quad | in^{13} \text{server}.give\_id^{14}[out^{15} p. < p >^{16}]] \\ &\quad | < \text{make} >^{17}) | < \text{make} >^{18}]) \end{aligned}$$

In this protocol, some packets are created (at program point **10**). They are initially located in the ambient **client**<sup>8</sup>[•]. Each packet is identified by a fresh name **p**. The packet contains some routing information to enter the ambient **server**<sup>1</sup>[•]. Once inside the server ambient, the packet expels an ambient **giveid**<sup>14</sup>[•] in order to communicate the name of the packet to the server. The server may open this ambient (at program point **2**), receive the name of the packet (at program point **3**), and create an ambient (at program point **4**) that enters the packet. Then the packet opens (at program point **12**) this ambient to receive the capability to return inside the ambient **client**<sup>8</sup>[•]. In this example, we abstract away what is happening to the packet while it is in the server domain.

We partition the threads (both agents and ambients) according to their location and the location of their surrounding ambient. Then, we count the number of threads inside each class of the partition. Our analysis discovers the contents of the packet according to its position in the network. For instance, we detect that whenever the packet is inside the ambient **client**<sup>8</sup>[•]: it contains only threads at the program points **4**, **11**, **12**, and **13**; moreover, either there is exactly one thread at each program point **11**, **12**, and **13**, or no threads at these three program points. Similar information are inferred for the other potential locations of the packet. We notice that our analysis loses all information about the number of threads at program point **4**, because it cannot infer that for a given packet, only one instance can receive the name of the packet. But since we detect that only one can be opened, this has no influence on the inference of the other properties. As in Ex. 4.1, we require an abstraction of the local history of each packet to reach this accuracy level: we count the number  $y_\lambda$  of each kind



$\lambda$  of transition, we also consider the variables  $z_\lambda$  that are defined as  $z_\lambda = 0$  if  $y_\lambda = 0$ , and  $z_\lambda = 1$  otherwise.  $\square$

## 5 Non-standard semantics

To prove the properties that interest us, we need to distinguish recursive instances of threads. Standard semantics are not convenient, because the  $\alpha$ -conversion breaks the relations between the threads and the name of the channels that they open. In this section, we recall a non-standard semantics [18, 21, 22]. This semantics is more concrete: each thread is annotated with information about both its history and the history of the names that it handles.

### 5.1 Notations

We consider a closed mobile system  $\mathcal{S}$  (i.e.  $\text{fv}(\mathcal{S}) = \emptyset$ ) in the  $\pi$ -calculus. We may assume that each variable is bound exactly once in the system (either by a name restriction or by an input). We may also assume that syntactic components are labeled with distinct labels. For any label  $l \in \mathcal{L}$ , we denote by  $\text{comp}(l)$  the subprocess the first action of which is labeled with  $l$ . We define  $\text{type}(l)$  as *input* if  $\text{comp}(l)$  matches  $c^?[x_1, \dots, x_n].Q$ , as *output* if  $\text{comp}(l)$  matches  $c^![x_1, \dots, x_n].Q$ , and as *fetch* if  $\text{comp}(l)$  matches  $*c^?[x_1, \dots, x_n].Q$ . Besides, with the same notations, we define  $\text{chan}(l) := c$ ,  $\text{arg}(l) := [x_1, \dots, x_n]$ , and  $\text{cont}(l) := Q$ . For any process  $P$ , we define the set  $\beta(P)$  of the labels of the threads that are launched in  $P$ ,  $\beta(P \mid Q) := \beta(P) \cup \beta(Q)$ ,  $\beta((\nu x)P) := \beta(P)$ ,  $\beta(\mathbf{0}) := \emptyset$ , and  $\beta(c^![x_1, \dots, x_n].P) := \beta(c^?[x_1, \dots, x_n].P) := \beta(*c^?[x_1, \dots, x_n].P) := \{l\}$ . For any label  $l$ , we denote by  $\text{I}(l)$  the set of the variables that are free in the threads at program point  $l$ . Thus, we define  $\text{I}(l)$  as  $\text{fv}(\text{comp}(l))$ .

### 5.2 Semantics

We define a non-standard semantics in which both threads and channel names are tagged with the history of their creation. History markers  $id \in \mathcal{L}^*$  are sequences of labels in  $\mathcal{L}$ . Markers encode the history of the replications which have led to the creation of thread instances. The markers of initial threads are  $\varepsilon$ . When a computation step does not involve fetching a resource, markers are just passed to the continuations; when a resource is fetched, the new instance is tagged with  $l_1.id_1$  where  $l_1$  and  $id_1$  are respectively the label and the marker of the output thread.

Then, we stamp each name with the marker of the thread which has declared it. Thus, a channel name is a pair  $(x, id)$  composed of a variable  $x \in \mathcal{V}$  and a marker  $id \in \mathcal{L}^*$ , which means that this is the name of the channel that has been opened by the restriction  $(\nu x)$  of a thread tagged with the marker  $id$ .

A *configuration* of the system  $\mathcal{S}$  is a set of *thread instances*. Each thread instance is a 3-tuple composed of a *label*  $l \in \mathcal{L}$  that denotes a syntactic component, an unambiguous *marker*  $id \in \mathcal{L}^*$ , and an *environment*  $E \in \text{I}(l) \rightarrow \mathcal{V} \times \mathcal{L}^*$  which

$$\begin{array}{c}
\mathcal{I} := \text{launch}(S, \varepsilon, \emptyset) \\
\text{(a) Non-standard initial configuration.} \\
\\
\frac{
\begin{cases}
E_?( \text{chan}(l_?) ) = E_l( \text{chan}(l_l) ), \\
\text{type}(l_?) = \text{input}, \text{type}(l_l) = \text{output}, \\
[y_1, \dots, y_n] := \text{arg}(l_?), [x_1, \dots, x_n] := \text{arg}(l_l), \\
Ct_? := \text{launch}(\text{cont}(l_?), id_?, E_?[y_k \mapsto E_l(x_k)]), \\
Ct_l := \text{launch}(\text{cont}(l_l), id_l, E_l),
\end{cases}
}{
C \cup \{ (l_?, id_?, E_?); (l_l, id_l, E_l) \} \xrightarrow{(l_?, l_l)} (C \cup Ct_? \cup Ct_l)
} \\
\\
\frac{
\begin{cases}
E_?( \text{chan}(l_?) ) = E_l( \text{chan}(l_l) ), \\
\text{type}(l_?) = \text{fetch}, \text{type}(l_l) = \text{output}, \\
[y_1, \dots, y_n] := \text{arg}(l_?), [x_1, \dots, x_n] := \text{arg}(l_l), \\
Ct_? := \text{launch}(\text{cont}(l_?), l_l.id_l, E_?[y_k \mapsto E_l(x_k)]), \\
Ct_l := \text{launch}(\text{cont}(l_l), id_l, E_l),
\end{cases}
}{
C \cup \{ (l_?, id_?, E_?); (l_l, id_l, E_l) \} \xrightarrow{(l_?, l_l)} (C \cup \{ (l_?, id_?, E_?) \} \cup Ct_? \cup Ct_l)
} \\
\text{(b) Non-standard transition system.}
\end{array}$$

Figure 1: Non-standard semantics.

specifies the channel names to which free variables are bound. Thread instances are created at the beginning of the computation and when agents interact. The function `launch` applied to a subprocess, a marker, and an environment, collects all the threads that are spawned when a continuation is launched: we set  $\text{launch}(P, id, E) := \{ (l, id, E_l) \mid l \in \beta(P) \}$ , where,  $E_l \in \mathbf{I}(l) \rightarrow \mathcal{V} \times \mathcal{L}^*$  maps any  $x \in \mathbf{I}(l) \cap \text{Dom}(E)$  to  $E(x)$ , and any  $x \in \mathbf{I}(l) \setminus \text{Dom}(E)$  to  $(x, id)$ . This simulates name restriction by binding any new variable  $x$  to the name of the channel opened by the restriction  $(\nu x)$  of a thread the marker of which is  $id$ . The initial state and computation rules are given in Fig. 1. The correspondence between the non-standard and the usual semantics is proved in [21, 22].

**Example 5.1 (the shared memory (cont.))** *We apply our non-standard semantics with our shared memory example (see. Ex. 4.1). We obtain the initial state  $C_0 = \{t_1; t_2; t_3\}$  where:*

$$\begin{aligned}
t_1 &= (1, \varepsilon, [\text{alloc} \mapsto (\text{alloc}, \varepsilon), \text{null} \mapsto (\text{null}, \varepsilon)]), \\
t_2 &= (12, \varepsilon, [\text{rec}_{12} \mapsto (\text{rec}_{12}, \varepsilon)]), \text{ and} \\
t_3 &= (12', \varepsilon, [\text{alloc} \mapsto (\text{alloc}, \varepsilon), \text{rec}_{12} \mapsto (\text{rec}_{12}, \varepsilon)]).
\end{aligned}$$

*The thread  $t_1$  is a resource that can allocate memory cells, the thread  $t_2$  can interact with the thread  $t_3$  to create clients recursively. Since these three threads are in the initial state, their thread marker is  $\varepsilon$ .*

We create a first client by making the threads  $t_2$  and  $t_3$  interact. We obtain the state  $C_1 = \{t_1; t_3; t_4; t_5\}$ , where:

$$t_4 = (12'', 12, [\text{rec}_{12} \mapsto (\text{rec}_{12}, \varepsilon)]) \text{ and} \\ t_5 = (13, 12, [\text{alloc} \mapsto (\text{alloc}, \varepsilon), \text{add} \mapsto (\text{add}, 12)]).$$

The thread  $t_4$  allows the creation of further clients and the thread  $t_5$  describes a client that can allocate a memory cell. We create a second client by making the threads  $t_4$  and  $t_3$  interact. We get the state  $C_2 = \{t_1; t_3; t_5; t_6; t_7\}$ , where:

$$t_6 = (12'', 12''.12, [\text{rec}_{12} \mapsto (\text{rec}_{12}, \varepsilon)]) \text{ and} \\ t_7 = (13, 12''.12, [\text{alloc} \mapsto (\text{alloc}, \varepsilon), \text{add} \mapsto (\text{add}, 12''.12)]).$$

The thread  $t_6$  allows the creation of further clients and the thread  $t_7$  describes the second client. Both clients are identified by their thread markers 12 and 12''.12. Besides, the link between threads and the channel names that they handle is explicit: the thread  $t_5$  can operate on the name  $(\text{add}, 12)$ , whereas the thread  $t_7$  can operate on the name  $(\text{add}, 12''.12)$ .  $\square$

## 6 Thread partitioning and trace partitioning

In this section, we first partition the threads of the system in several partition classes. Then we partition computation steps according to some relations about the threads that are involved. As a result, we obtain an extended labeled transition system.

### 6.1 Thread partitioning

Let  $B$  be a finite set of keys. Our analysis is parameterized by a function  $\text{getvar}$  mapping each program point label  $l$  to a function in  $B \rightarrow \mathbf{I}(l)$ . Then, we partition the threads  $t = (l, id, E)$  in a configuration according to the value  $E(\text{getvar}(l)(b))$  of the variable  $\text{getvar}(l)(b)$  for each key  $b \in B$ . We can also partition threads according to the markers of their names (we focus on partitioning according to full names to simplify the presentation.). For example, to prove the absence of race conditions, we gather the threads that operate on the same channel (we define  $B$  as  $\{b\}$  and  $\text{getvar}(l)(b)$  as  $\text{chan}(l)$ ). In *ambients*, we partition threads in accordance with their location and the location of their surrounding *ambient* (thus,  $B$  contains two keys). We know that: whenever two threads are in the same *ambient*, the location of their surrounding *ambient* is the same (partitioning the threads also according to the location of their surrounding *ambient* allows for a more precise partitioning at the abstract level). We denote by  $B_s \subseteq B$  a set of keys, such that: for any configuration  $C$ , for any threads  $t_1 = (p_1, id_1, E_1)$  and  $t_2 = (p_2, id_2, E_2)$  in the configuration  $C$ , if, for any  $b \in B_s$ ,  $E_1(\text{getvar}(p_1)(b)) = E_2(\text{getvar}(p_2)(b))$ , then, for any  $b \in B$ ,  $E_1(\text{getvar}(p_1)(b)) = E_2(\text{getvar}(p_2)(b))$ . This implication will be useful whenever we know that two threads are in the same configuration but in distinct partition classes.

Each partition class is identified by a function  $f \in B \rightarrow \mathcal{V} \times \mathcal{L}^*$ , called *computation unit*. We denote by  $\text{UNIT}$  the set  $B \rightarrow \mathcal{V} \times \mathcal{L}^*$  of all computation units. There may be an unbounded number of computation units. We gather them into a finite set of abstract computation units by abstracting away the information about markers: we define the set  $\text{UNIT}^\#$  of abstract computation units as  $B \rightarrow \mathcal{V}$ . The abstraction function  $\alpha_{\text{UNIT}}$  maps each computation unit  $[(b \in B) \mapsto (l_b, id_b)] \in \text{UNIT}$  to the abstract one  $[b \mapsto l_b] \in \text{UNIT}^\#$ .

## 6.2 Local trace partitioning

We consider a computation step  $\tau = (C \xrightarrow{(l_?, l_!)} C')$ . We denote by  $t_? = (l_?, id_?, E_?)$  and by  $t_! = (l_!, id_!, E_!)$  the threads that interact in the computation step  $\tau$ . The thread  $t_?$  launches one thread for each label  $l$  in the set  $\beta(\text{cont}(l_?))$  and the thread  $t_!$  launches one thread for each label  $l$  in the set  $\beta(\text{cont}(l_!))$ . We denote by  $n_?$  the cardinal of the set  $\beta(\text{cont}(l_?))$  and by  $n_!$  the cardinal of the set  $\beta(\text{cont}(l_!))$ . Thus, the computation step  $\tau$  involves  $2 + n_? + n_!$  threads. Each of these threads is denoted by a pair  $(l, \diamond) \in \mathcal{L} \times \{?; !\}$  where  $l$  is the label of the thread program point and  $\diamond$  is equal to  $?$  when this thread is related to the input thread or to  $!$  when this thread is related to the output thread. This way, we denote by  $\mathcal{T}(l_?, l_!)$  the set  $\{(l, \diamond) \mid \diamond \in \{?; !\}, l \in \{l_\diamond\} \cup \beta(\text{cont}(l_\diamond))\}$ .

To get a more precise analysis, we partition the set of computation steps according to some properties about the computation units of the threads that are involved in these computation steps. We denote by  $\text{CONTEXT}(l_?, l_!)$  the set of pairs  $(\sim, A)$  such that  $\sim$  is an equivalence relation<sup>2</sup> in  $\wp(\mathcal{T}(l_?, l_!))^2$  that relates the threads that share the same computation unit and  $A \in (\mathcal{T}(l_?, l_!))_\sim \rightarrow \text{UNIT}^\#$  maps each equivalence class to its abstract computation unit. Intuitively, the relation  $(l_1, \diamond_1) \sim (l_2, \diamond_2)$  means that the thread denoted by the pair  $(l_1, \diamond_1)$  and the thread denoted by the pair  $(l_2, \diamond_2)$  are in the same computation unit; moreover,  $A([(l, \diamond)]_\sim)$  is the abstract computation unit of the thread denoted by the pair  $(l, \diamond)$ . More formally, we denote by  $unit_\tau$  the function which maps any pair  $(l, \diamond) \in \mathcal{T}(l_?, l_!)$  to the computation unit of the thread denoted by the pair  $(l, \diamond)$ . Then, we define the abstraction function  $\alpha_{\text{STEP}}$  which maps each computation step to its partition case as:  $\alpha_{\text{STEP}}(\tau) := (\sim, [[a]_\sim \mapsto \alpha_{\text{UNIT}}(unit_\tau(a))])$ , where  $\sim$  is defined as  $a \sim b$  if and only if  $unit_\tau(a) = unit_\tau(b)$ .

## 7 Abstraction

In this section, we use the abstract interpretation framework [11, 13] to design a generic abstraction of transition systems.

<sup>2</sup>Given an equivalence relation  $\sim$  over a set  $A$ ,  $[a]_\sim$  denotes the equivalence class  $\{b \in A \mid a \sim b\}$  of  $a$  and  $A_\sim$  denotes the set  $\{[a]_\sim \mid a \in A\}$  of equivalence classes.

## 7.1 Reachable states

We denote by  $\mathcal{C}$  the set of all configurations, by  $\Sigma$  the set of pairs in  $\bigcup_{\lambda \in \mathcal{L}^2} \{\lambda\} \times \text{CONTEXT}(\lambda)$ , and, for any finite set  $V \subseteq \mathcal{V}$  of variables, by  $\mathcal{E}(V) := \mathcal{L}^* \times (V \rightarrow (\mathcal{V} \times \mathcal{L}^*))$  the set of marker/environment (over  $V$ ) pairs. We are interested in  $\mathcal{C}(\mathcal{S})$ , the set of all configurations that are reachable through a finite computation sequence. The set  $\mathcal{C}(\mathcal{S})$  is the least fixpoint of the  $\sqcup$ -complete endomorphism  $\mathbb{F}$  on the complete lattice  $\wp(\mathcal{C})$ , where  $\mathbb{F}$  is defined as  $[X \mapsto \mathcal{I} \cup \{\overline{C} \in \mathcal{C} \mid \exists C \in X, \exists \lambda \in \Sigma, C \xrightarrow{\lambda} \overline{C}\}]$ . This least fixpoint is usually not decidable, so we use a relaxed version of the abstract interpretation framework [15] to compute a sound—but not necessarily complete—approximation of it.

## 7.2 Generic abstraction

We choose an *abstract domain*, which is a set of abstract symbolic properties about configurations. It captures the properties of interest and abstracts away the other properties. Each abstract property is mapped to the set of the concrete elements which satisfy this property by a concretization map  $\gamma \in \mathcal{C}^\# \rightarrow \wp(\mathcal{C})$ . The abstract domain is fitted with several primitives to handle its elements. An abstract union  $\sqcup \in \wp_{\text{finite}}(\mathcal{C}^\#) \rightarrow \mathcal{C}^\#$  gathers the information described by several abstract elements. It satisfies:  $\forall a^\# \in A^\#, \gamma(a^\#) \subseteq \gamma(\sqcup(A^\#))$ . We also need an abstraction  $\mathcal{I}^\# \in \mathcal{C}^\#$  of the initial configuration (i.e.  $\mathcal{I} \in \gamma(\mathcal{I}^\#)$ ). To simulate computation steps in the abstract, we introduce an abstract operator  $\text{POST} \in \mathcal{C}^\# \times \Sigma \rightarrow \mathcal{C}^\#$ . This operator partitions each transition into several sub-cases: given an abstract property  $C^\# \in \mathcal{C}^\#$  and a sub-case  $\overline{\lambda} = (\lambda, \text{context}) \in \Sigma$ , the set  $\gamma(\text{POST}(C^\#, \overline{\lambda}))$  contains all the states  $\overline{C} \in \mathcal{C}$  that are reachable from any state  $C \in \gamma(C^\#)$  by a computation step  $\tau = C \xrightarrow{\lambda} \overline{C}$  such that  $\text{context} = \alpha_{\text{STEP}}(\tau)$ . An abstract element  $\perp$  such that  $\gamma(\perp) = \emptyset$  provides the basis for our abstract iteration. Finally, we use a widening operator  $\nabla : \mathcal{C}^\# \times \mathcal{C}^\# \rightarrow \mathcal{C}^\#$  to ensure the termination of our analysis. It satisfies  $\forall C_1^\#, C_2^\# \in \mathcal{C}^\#, \gamma(C_1^\#) \subseteq \gamma(C_1^\# \nabla C_2^\#)$  and  $\gamma(C_2^\#) \subseteq \gamma(C_1^\# \nabla C_2^\#)$ ; moreover, for any sequence  $(C_n) \in \mathcal{C}^{\mathbb{N}}$ , the sequence  $(C_n^\nabla)$  that is defined by  $C_0^\nabla := C_0$  and  $C_{n+1}^\nabla := C_n^\nabla \nabla C_{n+1}$  for any  $n \geq 0$ , is ultimately stationary. We do not use narrowing because, we iterate only functions  $f \in \mathcal{C}^\# \rightarrow \mathcal{C}^\#$  that satisfy:  $\gamma(a) \subseteq \gamma(f(a))$ .

**Definition 7.1** Any tuple  $(\mathcal{C}^\#, \sqcup, \perp, \gamma, \mathcal{I}^\#, \text{POST}, \nabla)$  that satisfies these assumptions is called an *abstraction*.

Given an abstraction  $\mathcal{A} = (\mathcal{C}^\#, \sqcup, \perp, \gamma, \mathcal{I}^\#, \text{POST}, \nabla)$ , we define the abstract counterpart  $\mathbb{F}_\mathcal{A}^\#$  of the function  $\mathbb{F}$  as the function that maps any abstract element  $C^\# \in \mathcal{C}^\#$  to the abstract element  $\sqcup(\{\text{POST}(C^\#, \overline{\lambda}) \mid \overline{\lambda} \in \Sigma\} \cup \{\mathcal{I}^\#\})$ . The function  $\mathbb{F}_\mathcal{A}^\#$  satisfies the soundness condition  $\forall C^\# \in \mathcal{C}^\#, \mathbb{F} \circ \gamma(C^\#) \subseteq \gamma \circ \mathbb{F}_\mathcal{A}^\#(C^\#)$ .

Then, we extrapolate the iterates of  $\mathbb{F}_\mathcal{A}^\#$ . We define the abstract iteration [15, 16] of  $\mathbb{F}_\mathcal{A}^\#$  as  $\mathcal{F}_0^\nabla := \perp$  and  $\mathcal{F}_{n+1}^\nabla := \mathcal{F}_n^\nabla \nabla \mathbb{F}_\mathcal{A}^\#(\mathcal{F}_n^\nabla)$  for any  $n \geq 0$ . The abstract iteration  $(\mathcal{F}_n^\nabla)_{n \in \mathbb{N}}$  is ultimately stationary. Moreover, its limit  $\llbracket \mathcal{S} \rrbracket_\mathcal{A}$  satisfies  $\mathcal{C}(\mathcal{S}) \subseteq \gamma(\llbracket \mathcal{S} \rrbracket_\mathcal{A})$  because  $\mathbb{F}$  is monotonic.

### 7.3 Coalesced product

Several abstractions can be composed to refine each other. We consider two abstractions:

$$\begin{aligned}\mathcal{A}_1 &= (\mathcal{C}_1^\sharp, \sqcup_1, \perp_1, \gamma_1, \mathcal{I}_1^\sharp, \text{POST}_1, \nabla_1), \text{ and} \\ \mathcal{A}_2 &= (\mathcal{C}_2^\sharp, \sqcup_2, \perp_2, \gamma_2, \mathcal{I}_2^\sharp, \text{POST}_2, \nabla_2).\end{aligned}$$

We define the coalesced product between the abstractions  $\mathcal{A}_1$  and  $\mathcal{A}_2$  as the tuple  $(\mathcal{C}^\sharp, \sqcup, \perp, \gamma, \mathcal{I}^\sharp, \text{POST}, \nabla)$ , where the domain  $\mathcal{C}^\sharp$  is defined as  $\mathcal{C}_1^\sharp \times \mathcal{C}_2^\sharp$ ; the concretization  $\gamma$  is defined as the intersection of the two concretizations (i.e.  $\gamma(a, b) := \gamma_1(a) \cap \gamma_2(b)$ ); the abstract union  $\sqcup$ , the element  $\perp$ , the widening operator  $\nabla$ , and the abstraction  $\mathcal{I}^\sharp$  of the initial state are all defined pairwise; the abstract element  $\text{POST}((C_1, C_2), \bar{\lambda})$  is defined as  $\perp$  whenever either  $\text{POST}_1(C_1, \bar{\lambda}) = \perp_1$  or  $\text{POST}_2(C_2, \bar{\lambda}) = \perp_2$ , and as  $(\text{POST}_1(C_1, \bar{\lambda}), \text{POST}_2(C_2, \bar{\lambda}))$  otherwise. The coalesced product between  $\mathcal{A}_1$  and  $\mathcal{A}_2$  is also an abstraction. We stress on the fact that the coalesced product is more powerful than a mere product. Thanks to the extended labeled transition system, several analyses can share constraints about the threads that are involved in computation steps. This way, analyses refine each other.

We use our framework with the coalesced product between an analysis of the dynamic linkage between threads (Sect. 8) and an analysis of each computation unit contents (Sect. 9).

## 8 Environment analysis

We design an analysis of the dynamic linkage between threads. This analysis aims at capturing the relationship between the computation units of the threads that are involved in computation steps.

### 8.1 Abstract domain

Our goal is to map each program point label to an abstraction of the set of the marker/environment pairs which may be associated to any thread at this program point at run-time. So, we introduce for any set of variables  $V \subseteq \mathcal{V}$  a parametric abstract domain  $\text{Atom}(V)$  of properties. The concretization  $\gamma_V(a)$  of a property  $a \in \text{Atom}(V)$  is a set of marker/environment pairs  $m$  in  $\wp(\mathcal{E}(V))$ . The operator  $\sqcup_V$  maps each finite set of properties to a weaker property: for each finite set  $A \subseteq \text{Atom}(V)$ ,  $\forall a \in A$ ,  $\gamma_V(a) \subseteq \gamma_V(\sqcup_V A)$ . The element  $\perp_V$  is an abstraction of the empty set (i.e. we assume that  $\gamma_V(\perp_V) = \emptyset$ ). The operator  $\nabla_V$  is a widening operator [16]. Then, our main environment abstract domain  $\mathcal{C}_{\text{env}}^\sharp$  is the set of the functions that map each program point label  $l \in \mathcal{L}$  that occurs in the system  $\mathcal{S}$  to an element in  $\text{Atom}(\mathcal{I}(l))$ . The domain structure  $(\sqcup_{\text{env}}, \perp_{\text{env}}, \text{and } \nabla_{\text{env}})$  is defined point wise. The abstract domain  $\mathcal{C}_{\text{env}}^\sharp$  is related to  $\wp(\mathcal{C})$  by the concretization function  $\gamma_{\text{env}}$  that maps each abstract property  $f \in \mathcal{C}_{\text{env}}^\sharp$  to the set of configurations  $C \in \mathcal{C}$  such that  $\forall (l, id, E) \in C$ ,  $(id, E) \in \gamma_{\mathcal{I}(l)}(f(l))$ .

**Example 8.1 (labels and equalities)** We propose a simple cfa domain to analyze the shared memory example (see Ex. 4.1). In this example, the names that occur in computation units are never communicated. As a consequence, equality among variables [20, Sect. 5.1.1] and a uniform approximation of the control flow [5, 4] are enough. In general, numerical abstractions of markers [18, 20] are required. All these analyses [5, 4, 20, 18] are polynomial time.

Given a set  $\mathcal{K}$  of variables, we introduce the abstract domain  $\mathcal{F}(\mathcal{K})$  as the set  $\perp_{\mathcal{F}(\mathcal{K})} \sqcup ((\mathcal{K} \mapsto \wp(\mathcal{L})) \times \wp(\mathcal{K} \times \{=, \neq\} \times \mathcal{K}))$ . Each abstract element  $e \in \mathcal{F}(\mathcal{K})$  denotes a set  $\gamma_{\mathcal{F}(\mathcal{K})}(e) \subseteq \mathcal{K} \mapsto \mathcal{L} \times \mathcal{L}^*$  of functions. More precisely,  $\gamma_{\mathcal{F}(\mathcal{K})}(\perp_{\mathcal{F}(\mathcal{K})}) = \emptyset$  and  $\gamma_{\mathcal{F}(\mathcal{K})}(f, c) = \{g \mid \forall (x, \diamond, y) \in c, g(x) \diamond g(y) \text{ and } \forall x \in \mathcal{K}, \exists id \in \mathcal{L}^*, g(x) = (f(x), id)\}$ . This way, in the abstract element  $(f, c)$ , the function  $f$  describes constraints about the label of values and the set  $c$  describes constraints about equality and inequality relations among values.

We define a partial order  $\sqsubseteq_{\mathcal{F}(\mathcal{K})}$  over  $\mathcal{F}(\mathcal{K})$  as:  $\perp \sqsubseteq_{\mathcal{F}(\mathcal{K})} x$ , for any  $x \in \mathcal{F}(\mathcal{K})$ , and  $(f_1, c_1) \sqsubseteq_{\mathcal{F}(\mathcal{K})} (f_2, c_2)$  if and only if both  $f_1(x) \subseteq f_2(x)$  and  $c_2 \subseteq c_1$ . We notice that the concretization  $\gamma_{\mathcal{F}(\mathcal{K})}$  is monotonic with respect to  $\sqsubseteq_{\mathcal{F}(\mathcal{K})}$ . Several abstract elements may have the same concretization, nevertheless, for each abstract element  $e \in \mathcal{F}(\mathcal{K})$ , the set of the elements  $e' \in \mathcal{F}(\mathcal{K})$  such that  $\gamma_{\mathcal{F}(\mathcal{K})}(e) = \gamma_{\mathcal{F}(\mathcal{K})}(e')$  has a least element that we denote  $\rho(e)$ . The element  $\rho(e)$  is called the normal form of  $e$ . We denote by  $\mathcal{F}_n(\mathcal{K})$  the set  $\{\rho(e) \mid e \in \mathcal{F}(\mathcal{K})\}$  of all normal forms. We denote by  $\sqsubseteq_{\mathcal{F}_n(\mathcal{K})}$  the restriction of  $\sqsubseteq_{\mathcal{F}(\mathcal{K})}$  to  $\mathcal{F}_n(\mathcal{K})$ . Each subset  $A \subseteq \mathcal{F}_n(\mathcal{K})$  has a least upper bound with respect to  $\sqsubseteq_{\mathcal{F}_n(\mathcal{K})}$ , that we denote by  $\sqcup_{\mathcal{F}_n(\mathcal{K})}$ .

The domain  $\mathcal{F}_n(V)$  is a good candidate for  $\text{Atom}(V)$ . We also set  $\gamma_V := [a \mapsto \mathcal{L}^* \times \gamma_{\mathcal{F}_n(V)}(a)]$ ,  $\sqcup_V := \sqcup_{\mathcal{F}_n(V)}$ , and  $\perp_V := \perp_{\mathcal{F}(V)}$ . Since  $\mathcal{F}_n(V)$  is a finite domain, we define the widening operator  $\nabla_V$  as  $a \nabla_V b := \sqcup_{\mathcal{F}_n(V)} \{a; b\}$ .  $\square$

Now, we simulate the non-standard semantics in the abstract.

## 8.2 Initial state

At the beginning of the concrete computation, the configuration contains one thread at each program point the label of which is in the set  $\beta(\mathcal{S})$ . Thread markers are  $\varepsilon$  and environments map each free variable  $x$  to the name  $(x, \varepsilon)$ . In the abstract, we require two primitives. First, the abstract property  $\varepsilon_\emptyset \in \text{Atom}(\emptyset)$  is the abstraction of the pair  $(\varepsilon, \emptyset)$ . This means that:  $\{(\varepsilon, \emptyset)\} \subseteq \gamma_\emptyset(\varepsilon_\emptyset)$ . Then, the primitive  $\nu^\sharp$  simulates name allocation. Let  $V$  be a set of variables and  $x \in \mathcal{V} \setminus V$  be a fresh variable. The primitive  $\nu_x^\sharp$  is a function in  $\text{Atom}(V) \rightarrow \text{Atom}(V \cup \{x\})$  and, for any abstract element  $a \in \text{Atom}(V)$ , the concretization  $\gamma_{V \cup \{x\}}(\nu_x^\sharp(a))$  contains at least all pairs  $(id, E) \in \mathcal{E}(V \cup \{x\})$  such that (i)  $(id, E|_V) \in \gamma_V(a)$ , (ii)  $E(x) = (x, id)$ , and (iii)  $\forall y \in V, E(y) \neq E(x)$ .

**Example 8.2 (labels and equalities (cont.))** In our simple cfa domain (see Ex. 8.1), the primitive  $\varepsilon_\emptyset$  can be defined as  $(\emptyset, \emptyset)$  (where, in the first component, the symbol  $\emptyset$  denotes the function defined over the empty set). Moreover, we define  $\nu_x^\sharp$  by:  $\nu_x^\sharp(\perp_V) := \perp_V$  and by  $\nu_x^\sharp((f, c)) := \rho(f', c')$  where  $f' := f[x \mapsto x]$

$$\mathcal{I}_{\text{env}}^{\#} := \left[ \begin{array}{l} l \notin \beta(\mathcal{S}) \mapsto \perp_{\mathbf{I}(l)}, \\ l \in \beta(\mathcal{S}) \mapsto \nu_{x_n}^{\#}(\dots(\nu_{x_1}^{\#}(\varepsilon_{\emptyset}))\dots), \text{ (where } \{x_1, \dots, x_n\} := \mathbf{I}(l)) \end{array} \right].$$

(a) Initial configuration abstraction.

Let  $l_?$  and  $l_l$  be two program point labels in  $\mathcal{L}$ , such that  $\mathbf{type}(l_?) \in \{\text{input}, \text{fetch}\}$ ,  $\mathbf{type}(l_l) = \text{output}$ , and such that the length of the lists  $\mathbf{arg}(l_?)$  and  $\mathbf{arg}(l_l)$  is the same. We denote  $[y_1, \dots, y_n] = \mathbf{arg}(l_?)$  and  $[x_1, \dots, x_n] = \mathbf{arg}(l_l)$ . Let  $(\sim, A) \in \text{CONTEXT}(l_?, l_l)$  be a partition case and  $\text{ENV} \in \mathcal{C}_{\text{env}}^{\#}$  be an abstract element.

We define:

- $\text{INPUT}_0 := \text{ENV}(l_?)$  and  $\text{OUTPUT}_0 := \text{ENV}(l_l)$ ;
  - $\text{INPUT}_1 := \begin{cases} \text{FETCH}(l_l, \text{INPUT}_0) & \text{whenever } \mathbf{type}(l_?) = \text{fetch}, \\ \text{INPUT}_0 & \text{otherwise;} \end{cases}$
  - $\text{INPUT}_3 := \nu_{u_o}^{\#}(\dots(\nu_{u_1}^{\#}(\text{NEW}_{y_n}(\dots(\text{NEW}_{y_1}(\text{INPUT}_1))\dots))\dots)$   
where  $\{u_1; \dots; u_o\} := (\bigcup \{\mathbf{I}(l) \mid l \in \beta(\mathbf{cont}(l_?))\}) \setminus \mathbf{fv}(\mathbf{cont}(l_?))$ ,
  - $\text{OUTPUT}_3 := \nu_{v_p}^{\#}(\dots(\nu_{v_1}^{\#}(\text{OUTPUT}_0))\dots)$ ,  
where  $\{v_1; \dots; v_p\} := (\bigcup \{\mathbf{I}(l) \mid l \in \beta(\mathbf{cont}(l_l))\}) \setminus \mathbf{fv}(\mathbf{cont}(l_l))$ ;
  - $\text{mol}_0 := \text{INPUT}_3 \bullet \text{OUTPUT}_3$ ;
  - $\text{cons} := \text{com} \cup \text{part}_= \cup \text{part}_{\neq} \cup \text{part}_{\text{lbl}}$ , where:
    - $\text{com} := \{(\mathbf{chan}(l_?), ?) = (\mathbf{chan}(l_l), !)\} \cup \{(y_k, ?) = (x_k, !)\} \mid 1 \leq k \leq n\}$ ,
    - $\text{part}_= := \{(\mathbf{getvar}(l_1)(b), \diamond_1) = (\mathbf{getvar}(l_2)(b), \diamond_2) \mid (l_1, \diamond_1), (l_2, \diamond_2) \in \mathcal{T}(l_?, l_l), b \in B, (l_1, \diamond_1) \sim (l_2, \diamond_2)\}$ ,
    - $\text{part}_{\neq} := \{(\mathbf{getvar}(l_1)(b), \diamond_1) = (\mathbf{getvar}(l_2)(b), \diamond_2) \mid (l_1, \diamond_1), (l_2, \diamond_2) \in \mathcal{T}(l_?, l_l), (l_1, \diamond_1) \not\sim (l_2, \diamond_2), B_s = \{b\}\}$ ,
    - $\text{part}_{\text{lbl}} := \{\text{lbl}((\mathbf{getvar}(l)(b), \diamond), A([(l, \diamond)]_{\sim})(b)) \mid (l, \diamond) \in \mathcal{T}(l_?, l_l)\}$ ;
  - $\text{mol}_1 := \text{SYNC}(\text{cons}, \text{mol}_0)$ ;
  - $\text{POST}_{\text{env}}(\text{ENV}, ((l_?, l_l), (\sim, A))) := \begin{cases} \perp_{\text{env}} & \text{if } \text{mol}_1 = \perp_{(V_?, V_l)}, \\ \sqcup_{\text{env}} \{\text{ENV}; \text{ENV}'\} & \text{otherwise,} \end{cases}$
- where  $\text{ENV}' := \begin{cases} l \mapsto \text{GC}(\mathbf{I}(l), \text{FST}(\text{mol}_1)) & \text{whenever } l \in \beta(\mathbf{cont}(l_?)), \\ l \mapsto \text{GC}(\mathbf{I}(l), \text{SND}(\text{mol}_1)) & \text{whenever } l \in \beta(\mathbf{cont}(l_l)), \\ l \mapsto \perp_{\mathbf{I}(l)} & \text{otherwise.} \end{cases}$
- (b) Abstract POST operator.

Figure 2: Environment analysis.



and  $c' := c \cup \{(x, \neq, a) \mid a \in V\}$ . This means that we know that the channel has been opened by an instance of a restriction  $(\nu x)$  and we know that this value is fresh. Then, we apply our closure  $\rho$ .  $\square$

The abstraction  $\mathcal{I}_{\text{env}}^\# \in \mathcal{C}_{\text{env}}^\#$  of initial state is defined in Fig. 2(a) as the function that maps any program point  $l$  to the abstract element  $\nu_{x_n}^\# (\dots (\nu_{x_1}^\# (\varepsilon_\emptyset)) \dots)$  whenever  $l \in \beta(\mathcal{S})$  and  $\{x_1; \dots; x_n\} := \mathbf{I}(l)$ ; and to the abstract element  $\perp_{\mathbf{I}(l)}$  otherwise.

**Example 8.3 (the shared memory (cont.))** *We apply our analysis with the simple cfa abstract domain (e.g. see Ex. 8.1) on the shared memory system (e.g. see Ex. 4.1). We obtain that:  $\mathcal{I}_{\text{env}}^\#(1) = \rho([\text{alloc} \mapsto \text{alloc}, \text{null} \mapsto \text{null}], \emptyset)$ ,  $\mathcal{I}_{\text{env}}^\#(12) = \rho([\text{rec}_{12} \mapsto \text{rec}_{12}], \emptyset)$ ,  $\mathcal{I}_{\text{env}}^\#(12') = \rho([\text{alloc} \mapsto \text{alloc}, \text{rec}_{12} \mapsto \text{rec}_{12}], \emptyset)$ , and  $\mathcal{I}_{\text{env}}^\#(l) = \perp_{\mathbf{I}(l)}$  for any  $l \notin \{1; 12; 12'\}$ .  $\square$*

### 8.3 Transition step

In the concrete, an interaction involves two threads:  $t_?$  at a program point labeled with  $l_?$  and  $t_!$  at a program point labeled with  $l_!$ . The first thread either inputs a message or fetches a resource; the second thread outputs a message. We simulate such a transition  $\tau$  in the abstract in Fig. 2(b). We start from the abstract element  $\text{ENV} \in \mathcal{C}_{\text{env}}^\#$  and we define the pair  $(\sim, A) \in \text{CONTEXT}(l_?, l_!)$  as  $\alpha_{\text{STEP}}(\tau)$ .

**Example 8.4 (the shared memory (cont.))** *We apply our analysis with the simple cfa abstract domain (e.g. see Ex. 8.1) on the shared memory system (e.g. see Ex. 4.1). As an example, we focus on the interaction between a thread at program point **5** and a thread at program point **10** in any calling context  $(\sim, A) \in \text{CONTEXT}(l_?, l_!)$ . We also assume that the element  $\text{ENV}(5)$  is equal to  $\rho([\text{cell} \mapsto \{\text{cell}\}, \text{fwd} \mapsto \{\text{return}\}], \emptyset)$  and that the element  $\text{ENV}(10)$  is equal to  $\rho([\text{cell} \mapsto \{\text{cell}\}, \text{val}' \mapsto \{\text{data}\}], \emptyset)$ .*

*We want to prove that:*

- *both that interact and the thread that is launched at program point **6** belong to the same partition class (i.e.  $(5, ?) \sim (6, ?)$ ,  $(5, ?) \sim (10, !)$ );*
- *the thread that interacts at the program point **5** and the thread that is launched at the program point **7** do not belong to the same partition class (i.e.  $(5, ?) \not\sim (7, ?)$ );*
- *and that the abstraction of the computation unit of interacting threads is  $[b \mapsto \text{cell}]$  (i.e.  $A([(5, ?)]_\sim)(b) = \text{cell}$ ).*

*Then, we want to abstract the environment of the thread that is launched at program point **6**.  $\square$*

### 8.3.1 Extending environments

First, we collect information about the potential binding of the threads  $t_?$  and  $t_!$ . We denote by  $\text{INPUT}_0$  the element  $\text{ENV}(l_?)$  and by  $\text{OUTPUT}_0$  the element  $\text{ENV}(l_!)$ . In the concrete, a new thread marker is computed when the input thread is a resource. We require a primitive  $\text{FETCH}$  to simulate the allocation of this fresh marker in the abstract. For any set  $V \subseteq \mathcal{V}$  of variables, any abstract element  $a \in \text{Atom}(V)$ , and any label  $l \in \mathcal{L}$ , the abstract element  $\text{FETCH}(l, a) \in \text{Atom}(V)$  satisfies: the concretization  $\gamma_V(\text{FETCH}(l, a))$  contains at least all pairs  $(l.id, E) \in \mathcal{E}(V)$  such that  $(id, E) \in \gamma_V(a)$ .

**Example 8.5 (labels and equalities (cont.))** *In our simple cfa domain (see Ex. 8.1), we do not track any information about thread markers. So we define the element  $\text{FETCH}(l, a)$  as  $a$ .  $\square$*

Then, we define  $\text{INPUT}_1$  as  $\text{FETCH}(l_!, \text{INPUT}_0)$  whenever the thread  $t_?$  is a resource (i.e. if  $\text{type}(l_?) = \text{fetch}$ ), and as  $\text{INPUT}_0$  otherwise.

**Example 8.6 (the shared memory (cont.))** *In our example, we have:*

$$\begin{aligned} \text{INPUT}_1 &= \rho([\text{cell} \mapsto \{\text{cell}\}, \text{fwd} \mapsto \{\text{return}\}], \emptyset) \text{ and} \\ \text{OUTPUT}_0 &= \rho([\text{cell} \mapsto \{\text{cell}\}, \text{val} \mapsto \{\text{data}\}], \emptyset). \end{aligned}$$

$\square$

We now extend the environments to deal with the variables introduced during the interaction. In the concrete, the threads  $t_?$  and  $t_!$  bind some new variables to some names. The sequence  $[y_1, \dots, y_n] := \text{arg}(l_?)$  is the sequence of the variables that are bound by name passing. We use an abstract primitive  $\text{NEW}$  to create these variables without any information about them. For any set  $V \subseteq \mathcal{V}$  of variables, any variable  $x \notin V$ , and any abstract element  $a \in \text{Atom}(V)$ , the abstract element  $\text{NEW}_x(a) \in \text{Atom}(V \cup \{x\})$  satisfies:  $\{(id, E) \in \mathcal{E}(V \cup \{x\}) \mid (id, E|_V) \in \gamma_V(a)\} \subseteq \gamma_{V \cup \{x\}}(\text{NEW}_x(a))$ .

**Example 8.7 (labels and equalities (cont.))** *We can define the primitive  $\text{NEW}$  by  $\text{NEW}_x(\perp_V) := \perp_V$  and by  $\text{NEW}_x(f, c) := (f[x \mapsto \mathcal{L}], c)$ .  $\square$*

Thus, we define  $\text{INPUT}_2$  by  $\text{NEW}_{y_n}(\dots(\text{NEW}_{y_1}(\text{INPUT}_1))\dots)$ . The set of the variables that are bound by name restriction in the thread  $t_?$  is given by  $\{u_1; \dots; u_o\} := (\bigcup \{I(l) \mid l \in \beta(\text{cont}(l_?))\} \setminus \text{fv}(\text{cont}(l_?)))$ , whereas the one in the thread  $t_!$  is given by  $\{v_1; \dots; v_p\} := (\bigcup \{I(l) \mid l \in \beta(\text{cont}(l_!))\} \setminus \text{fv}(\text{cont}(l_!)))$ . We introduce these variables thanks to the primitive  $\nu^\sharp$ . We define  $\text{INPUT}_3 := \nu_{u_o}^\sharp(\dots(\nu_{u_1}^\sharp(\text{INPUT}_2))\dots)$  and  $\text{OUTPUT}_3 := \nu_{v_p}^\sharp(\dots(\nu_{v_1}^\sharp(\text{OUTPUT}_0))\dots)$ .

**Example 8.8 (the shared memory (cont.))** *In the shared memory example, the variable  $\text{val}$  is bound during the communication. Moreover, since no variable is bound by a name restriction, the abstract element  $\text{INPUT}_3$  is equal to  $\rho(f, \emptyset)$  where  $f = [\text{cell} \mapsto \{\text{cell}\}, \text{fwd} \mapsto \{\text{return}\}, \text{val} \mapsto \mathcal{L}]$ , and the abstract element  $\text{OUTPUT}_3$  is equal to  $\rho([\text{cell} \mapsto \{\text{cell}\}, \text{val} \mapsto \{\text{data}\}], \emptyset)$ .  $\square$*

To get precise relations between the binding of former variables and the binding of the variables bound by the communication, we gather the two descriptions  $\text{INPUT}_3$  and  $\text{OUTPUT}_3$ . For that purpose, we assume that we are given, for any subset of variables  $V_?, V_! \subseteq \mathcal{V}$ , an abstract domain  $\text{Molecule}(V_?, V_!)$  of properties about sets of pairs of marker/environment pairs. Each property in  $\text{Molecule}(V_?, V_!)$  is related by a concretization function  $\gamma_{(V_?, V_!)}$  to the elements of  $\wp(\mathcal{E}(V_?) \times \mathcal{E}(V_!))$  which satisfy this property. We also introduce an element  $\perp_{(V_?, V_!)}$  that satisfies  $\gamma_{(V_?, V_!)}(\perp_{(V_?, V_!)}) = \emptyset$ . The domains  $\text{Atom}(V_?)$ ,  $\text{Atom}(V_!)$ , and  $\text{Molecule}(V_?, V_!)$  are related by the following primitives. The primitive  $\bullet$  simulates pair construction. For any  $a_? \in \text{Atom}(V_?)$  and any  $a_! \in \text{Atom}(V_!)$ , the element  $a_? \bullet a_! \in \text{Molecule}(V_?, V_!)$  satisfies:  $\gamma_{V_?}(a_?) \times \gamma_{V_!}(a_!) \subseteq \gamma_{(V_?, V_!)}(a_? \bullet a_!)$ ; the primitives FST and SND abstract the projection functions: for any  $a \in \text{Molecule}(V_?, V_!)$ , the elements  $\text{FST}(a) \in \text{Atom}(V_?)$  and  $\text{SND}(a) \in \text{Atom}(V_!)$  satisfy:  $\gamma_{(V_?, V_!)}(a) \subseteq \gamma_{V_?}(\text{FST}(a)) \times \gamma_{V_!}(\text{SND}(a))$ .

Then, we gather the two properties thanks to the abstract product  $\bullet$ . We define  $\text{mol}_0$  as  $\text{INPUT}_3 \bullet \text{OUTPUT}_3$ . We denote by  $(V_?, V_!) \in \wp(\mathcal{V})^2$  the pair of sets of variables such that  $\text{mol}_0 \in \text{Molecule}(V_?, V_!)$ . The element  $\text{mol}_0$  abstracts a set of pairs  $((id_?, E_?), (id_!, E_!)) \in \mathcal{E}(V_?) \times \mathcal{E}(V_!)$ . We introduce some formal variable to denote the channel names that are bound either in the environment  $E_?$ , or in the environment  $E_!$ . We introduce the set  $\text{Var}(V_?, V_!) := \{(v, ?) \mid v \in V_?\} \cup \{(v, !) \mid v \in V_!\}$  of formal variables.

**Example 8.9 (labels and equalities (cont.))** *We can define the abstract domain  $\text{Molecule}(V_?, V_!)$  as  $\mathcal{F}_n(\text{Var}(V_?, V_!))$ . The concretization  $\gamma_{(V_?, V_!)}$  maps each abstract element  $a$  to the set of pairs  $(id_?, E_?), (id_!, E_!)$  such that the map  $[(x, ?) \mapsto E_?(x), (x, !) \mapsto E_!(x)]$  belongs to  $\gamma_{\mathcal{F}_n(\text{Var}(V_?, V_!))}(a)$ . The bottom element  $\perp_{(V_?, V_!)}$  can be defined as  $\perp_{\mathcal{F}(\text{Var}(V_?, V_!))}$ .*

*The primitive FST maps  $\perp_{(V_?, V_!)}$  to  $\perp_{V_?}$  and any other element  $(f, c)$  to the element  $[(x \in V_? \mapsto f(x, ?)), \{(x, \diamond, y) \mid ((x, ?), \diamond, (y, ?)) \in c\}]$ . The primitive SND maps  $\perp_{(V_?, V_!)}$  to  $\perp_{V_!}$  and any other element  $(f, c)$  to the element  $[(x \in V_! \mapsto f(x, !)), \{(x, \diamond, y) \mid ((x, !), \diamond, (y, !)) \in c\}]$ . The abstract product is defined by:  $\perp_{V_?} \bullet e_! = e_? \bullet \perp_{V_!} = \perp_{(V_?, V_!)}$  and by  $(f_?, c_?) \bullet (f_!, c_!) := (f', c')$ , where  $f' := [(x, i) \mapsto f_i(x)]$  and  $c' := \{((x, i), \diamond, (y, i)) \mid (x, \diamond, y) \in c_i\}$ .  $\square$*

**Example 8.10 (the shared memory (cont.))** *In our example, the abstract element  $\text{mol}_0$  is equal to  $\rho(f, \emptyset)$  where the function  $f$  is defined as the following function:*

$$\begin{cases} (\text{cell}, ?) \mapsto \{\text{cell}\}, \\ (\text{fwd}, ?) \mapsto \{\text{return}\}, \\ (\text{val}, ?) \mapsto \mathcal{L}, \\ (\text{cell}, !) \mapsto \{\text{cell}\}, \\ (\text{val}', !) \mapsto \{\text{data}\}. \end{cases}$$

$\square$

### 8.3.2 Collecting new constraints

Now, we collect the set  $cons$  of all the constraints that we have about the environments  $E_?$  and  $E_!$ . The formal variable  $(v, ?)$  denotes the value  $\sigma(v, ?) := E_?(v)$  of the variable  $v$  in the input thread and the variable  $(v, !)$  denotes the value  $\sigma(v, !) := E_!(v)$  of the variable  $v$  in the output thread. We consider three kinds of constraints: the constraint  $v_1 = v_2$  where  $v_1, v_2 \in Var(V_?, V_!)$  means that the formal variables  $v_1$  and  $v_2$  denote the same channel name: we write  $p \models v_1 = v_2$  if and only if  $\sigma(v_1) = \sigma(v_2)$ ; the constraint  $v_1 \neq v_2$  is the negation of the constraint  $v_1 = v_2$ : we write  $p \models v_1 \neq v_2$  if and only if  $\sigma(v_1) \neq \sigma(v_2)$ ; the constraint  $lbl(v, l)$ , where  $v \in Var(V_?, V_!)$  and  $l \in \mathcal{L}$  means that  $l$  is the label of the name that is denoted by the formal variable  $v$ : we write  $p \models lbl(v, l)$  if and only if  $\sigma(v)$  matches  $(l, \_)$ . We denote by  $Constraints(V_?, V_!)$  the set of all such constraints. First, we collect the constraints due to communication: the constraint  $(\mathbf{chan}(l_?), ?) = (\mathbf{chan}(l_!), !)$  encodes the fact that both threads interact over the same channel and the set  $\{(y_k, ?) = (x_k, !) \mid 1 \leq k \leq n\}$  of constraints encodes name-passing. Now we consider the constraints given by  $\sim$ : for any pair  $(l_1, \diamond_1), (l_2, \diamond_2) \in \mathcal{T}(l_?, l_!)$  such that  $(l_1, \diamond_1) \sim (l_2, \diamond_2)$ , the set of constraints  $\{(\mathbf{getvar}(l_1)(b), \diamond_1) = (\mathbf{getvar}(l_2)(b), \diamond_2) \mid b \in B\}$  encodes the fact that the threads that are denoted by the pairs  $(l_1, \diamond_1)$  and  $(l_2, \diamond_2)$  share the same computation unit; conversely, when  $B_s$  is not a singleton, we cannot extract constraints from non-equality among computation units, but when  $B_s$  is a singleton  $\{b\}$ , for any pairs  $(l_1, \diamond_1), (l_2, \diamond_2) \in \mathcal{T}(l_?, l_!)$  such that  $(l_1, \diamond_1) \not\sim (l_2, \diamond_2)$ , the constraint  $(\mathbf{getvar}(l_1)(b), \diamond_1) \neq (\mathbf{getvar}(l_2)(b), \diamond_2)$  encodes the fact that the threads that are denoted by the pairs  $(l_1, \diamond_1)$  and  $(l_2, \diamond_2)$  are not in the same computation unit; last, for any pair  $(l, \diamond) \in \mathcal{T}(l_?, l_!)$ , the set of constraints  $\{lbl(\mathbf{getvar}(l)(b), \diamond), A([(l, \diamond)]_\sim)(b)) \mid b \in B\}$  models the fact that  $A([(l, \diamond)]_\sim)$  is the abstract computation unit of the thread denoted by the pair  $(l, \diamond)$ .

**Example 8.11 (the shared memory (cont.))** *In our example, we get the constraint set  $com \cup part_{=} \cup part_{\neq} \cup part_{lbl}$ , where:*

$$com = \{(\mathbf{cell}, ?) = (\mathbf{cell}, !); (val, ?) = (val', !)\},$$

*and  $part_{=}$ ,  $part_{\neq}$ , and  $part_{lbl}$  are defined as in Fig. 2(b) (they depend on the pair  $(\sim, A)$ ).  $\square$*

We can now define  $mol_1$  as  $\text{SYNC}(cons, mol_0)$ , where the primitive  $\text{SYNC}$  is used to enforce some constraints in abstract elements. For any set  $C \in Constraints(V_?, V_!)$  of constraints and any abstract element  $a \in Molecule(V_?, V_!)$ , the element  $\text{SYNC}(C, a) \in Molecule(V_?, V_!)$  is such that the set  $\gamma_{(V_?, V_!)}(\text{SYNC}(C, a))$  contains at least all pairs  $p = ((id_?, E_?), (id_!, E_!))$  that satisfy both  $p \in \gamma_{(V_?, V_!)}(a)$  and  $\forall c \in C, p \models c$ .

**Example 8.12 (labels and equalities (cont.))** *We can define the primitive*

SYNC as follows:

$$\begin{cases} \text{SYNC}(\text{cons}, \perp_{(v_?, v_?)}) := \perp_{(v_?, v_?)}, \\ \text{SYNC}(\text{cons}, (f, c)) := \rho(f', c \cup \{(x, \diamond, y) \mid x \diamond y \in \text{cons}\}), \end{cases}$$

$$\text{where } f' = \begin{cases} x \mapsto f(x) & \text{whenever } \nexists l, \text{lbl}(v, l) \in \text{cons}, \\ x \mapsto \{l\} & \text{whenever } \exists l, \text{lbl}(v, l) \in \text{cons}, \\ x \mapsto \emptyset & \text{otherwise;} \end{cases}$$

We stress that the normalization step is crucial to propagate information, and especially to detect unsatisfiable constraints.  $\square$

**Example 8.13 (the shared memory (cont.))** First, we prove that the interaction is not possible as soon as  $(5, ?) \not\sim (6, ?)$ ,  $(5, ?) \not\sim (10, !)$ ,  $(5, ?) \sim (7, ?)$ , or  $A([(5, ?)]_{\sim})(b) \neq \text{cell}$ :

- If  $(5, ?) \not\sim (6, ?)$ , we have  $(\text{cell}, ?) \neq (\text{cell}, ?) \in \text{part}_{\neq}$ . Then,  $\text{mol}_1 = \perp_{(v_?, v_?)}$ .
- If  $(5, ?) \not\sim (10, !)$ , we have  $(\text{cell}, ?) \neq (\text{cell}, !) \in \text{part}_{\neq}$ . But  $(\text{cell}, ?) = (\text{cell}, !) \in \text{com}$ . Then,  $\text{mol}_1 = \perp_{(v_?, v_?)}$ .
- If  $(5, ?) \sim (7, ?)$ , we have  $(\text{cell}, ?) \sim (\text{fwd}, ?) \in \text{part}_{=}$ . Then,  $\text{mol}_1$  matches  $\rho(f, c)$  with  $(\text{cell}, ?) \sim (\text{fwd}, ?) \in c$ ,  $f(\text{cell}, ?) = \{\text{cell}\}$ , and  $f(\text{fwd}, ?) = \{\text{return}\}$ . Since  $f(\text{cell}, ?) \cap f(\text{fwd}, ?) = \emptyset$ , we have  $\text{mol}_1 = \perp_{(v_?, v_?)}$ .
- If  $A([(5, ?)]_{\sim})(b) \neq \text{cell}$ ,  $\{\text{lbl}(\text{cell}, ?), A([(5, ?)]_{\sim})(b)\} \in \text{part}_{\text{lbl}}$ . Then  $\text{mol}_1$  matches  $\rho(f, c)$  with  $f(\text{cell}, ?) = \{\text{cell}\} \cap \{A([(5, ?)]_{\sim})(b)\}$ . So  $\text{mol}_1 = \perp_{(v_?, v_?)}$ .

Until the end of the section, we assume that:  $(5, ?) \sim (6, ?)$ ,  $(5, ?) \sim (10, !)$ ,  $(5, ?) \not\sim (7, ?)$ , and  $A([(5, ?)]_{\sim})(b) = \text{cell}$ .

With these assumptions, we have:

- $\text{com} = \{(\text{cell}, ?) = (\text{cell}, !); (\text{val}, ?) = (\text{val}', !)\}$ ,
- $\text{part}_{=} = \{(\text{cell}, ?) = (\text{cell}, !); (\text{cell}, ?) = (\text{cell}, ?)\}$ ,
- $\text{part}_{\neq} = \{(\text{cell}, ?) \neq (\text{fwd}, ?)\}$ ,
- $\text{part}_{\text{lbl}} = \left\{ \begin{array}{l} \text{lbl}(\text{cell}, ?), \text{cell}; \text{lbl}((\text{cell}, !), \text{cell}); \\ \text{lbl}((\text{fwd}, ?), A([(7, ?)]_{\sim})) \end{array} \right\}$ .

Then,  $\text{mol}_1 = \rho(f, \text{com} \cup \text{part}_{=} \cup \text{part}_{\neq})$  where the function  $f$  is defined as

$$\begin{cases} (\text{cell}, ?) \mapsto \{\text{cell}\}, \\ (\text{fwd}, ?) \mapsto \{\text{return}\}, \\ (\text{val}, ?) \mapsto \mathcal{L}, \\ (\text{cell}, !) \mapsto \{\text{cell}\}, \\ (\text{val}', !) \mapsto \{\text{data}\}. \end{cases}$$

Since the constraint  $(val', ! ) = (val, ?)$  belongs to the set  $com$  of constraints, we can deduce that the abstract element  $mol_1$  is equal to  $\rho(f', com \cup part_{=} \cup part_{\neq})$ , where  $f' = f[(val, ?) \mapsto \{data\}]$ .  $\square$

### 8.3.3 Updating the abstract element

Whenever we have  $mol_1 = \perp_{(V_?, V_!)}$ , the constraints are not satisfiable, so we set  $POST_{env}(ENV, ((l_?, l_!), (\sim, A))) := \perp_{env}$ . Otherwise, we first separate information about the input and the output threads, then we update the information about the threads that are launched. For that purpose, we use a primitive GC to simulate garbage collection: for any sets  $X, V$  of variables such that  $X \subseteq V$ , and any abstract element  $a \in Atom(V)$ , the abstract element  $GC_X(a) \in Atom(X)$  satisfies the property  $\{(id, E|_X) \in \mathcal{E}(X) \mid (id, E) \in \gamma_V(a)\} \subseteq \gamma_X(GC_X(a))$ .

**Example 8.14 (labels and equalities (cont.))** *The primitive GC can be defined by  $GC_X(\perp_V) := \perp_X$  and by  $GC_X(f, c) := (f|_X, c \cap X \times \{=, \neq\} \times X)$ .  $\square$*

We define the element  $POST_{env}(ENV, ((l_?, l_!), (\sim, A)))$  by  $\sqcup_{env}\{ENV; ENV'\}$ , where  $ENV'(l) := GC(I(l), FST(mol_1))$  whenever the label  $l$  belongs to the set  $\beta(\text{cont}(l_?))$ ,  $ENV'(l) := GC(I(l), SND(mol_1))$  whenever the label  $l$  is in the set  $\beta(\text{cont}(l_!))$ , and  $ENV'(l) := \perp_{I(l)}$  otherwise.

**Example 8.15 (the shared memory (cont.))** *In our example, the function  $ENV'$  satisfies:  $ENV'(6)$  is equal to the element  $\rho(f, \emptyset)$ , where  $f = [(cell, ?) \mapsto \{cell\}, (val, ?) \mapsto \{val'\}]$ . This is a precise abstraction of the environment of the thread that is launched at the program point 6.*

## 8.4 Soundness

Thm. 8.16 states the soundness of our environment analysis.

**Theorem 8.16**  $(\mathcal{C}_{env}^\#, \sqcup_{env}, \perp_{env}, \gamma_{env}, \mathcal{I}_{env}^\#, POST_{env}, \nabla_{env})$  is an abstraction.

## 9 Contents analysis

Contents analysis counts both the number of threads inside each computation unit and the number of computation steps in the history of computation units. Its main goal is to detect mutual exclusion of threads inside computation units.

### 9.1 Abstract domain

Let  $\mathcal{K}$  be the set of variables  $\{x_l \mid l \in \mathcal{L}\} \cup \{y_\lambda \mid \lambda \in \mathcal{L}^2\} \cup \{z_\lambda \mid \lambda \in \mathcal{L}^2\}$ . We use these variables to abstract both the contents and the history of computation units. Given a computation unit: the variable  $x_l$  counts the occurrence number of threads at the program point  $l$  in this computation unit, the variable  $y_\lambda$  counts the number of computation steps labeled with  $\lambda$  that have modified this

computation unit, and the variable  $z_\lambda$  is equal to 1 if at least one computation step labeled with  $\lambda$  has modified the contents of this computation unit and is equal to 0 otherwise.

We assume that we are given an abstract domain  $\mathcal{N}(\mathcal{K})$  to abstract functions in  $\mathcal{K} \rightarrow \mathbb{N}$ . Each abstract property is related to the set  $\wp(\mathcal{K} \rightarrow \mathbb{N})$  by a concretization  $\gamma_{\mathcal{N}(\mathcal{K})}$ . An operator  $\sqcup_{\mathcal{N}(\mathcal{K})}$  maps each finite set of properties to a weaker property: for each finite set  $A \subseteq \wp(\mathcal{N}(\mathcal{K}))$ ,  $\forall a \in A$ ,  $\gamma_{\mathcal{N}(\mathcal{K})}(a) \subseteq \gamma_{\mathcal{N}(\mathcal{K})}(\sqcup_{\mathcal{N}(\mathcal{K})} A)$ . The element  $\perp_{\mathcal{N}(\mathcal{K})}$  is the abstraction of the empty set (i.e. we have  $\gamma_{\mathcal{N}(\mathcal{K})}(\perp_{\mathcal{N}(\mathcal{K})}) = \emptyset$ ). The operator  $\nabla_{\text{con}}$  is a widening [16]. Then, our main abstract domain  $\mathcal{C}_{\text{con}}^\#$  is the set  $\text{UNIT}^\# \rightarrow \mathcal{N}(\mathcal{K})$  of the functions mapping each abstract computation unit to an abstraction of its contents. The structure  $(\sqcup_{\text{con}}, \perp_{\text{con}}, \text{and } \nabla_{\text{con}})$  is defined point wise. We define the concretization  $\gamma_{\text{con}}(\text{CU})$  of any abstract element  $\text{CU} \in \mathcal{C}_{\text{con}}^\#$  as the set of all configurations  $C \in \mathcal{C}$  such that for any concrete computation unit  $u \in \text{UNIT}$ ,  $\text{CU}(\alpha_{\text{UNIT}}(u))$  is an approximation of the contents of  $u$ . More precisely, we require that there exists a map  $n \in \gamma_{\text{con}}(\text{CU}(\alpha_{\text{UNIT}}(u)))$  such that  $\forall l \in \mathcal{L}$ , the number of threads in  $C$  at the program point  $l$  in the computation unit  $u$  is equal to  $n(x_l)$ . We also require that, for any  $\lambda \in \mathcal{L}^2$ , we have  $n(z_\lambda) = 1$  whenever  $n(y_\lambda) \geq 1$ , and  $n(z_\lambda) = 0$  otherwise (we require no further properties about the variables  $y_\lambda$  and  $z_\lambda$ ).

**Example 9.1 (interval and affine constraints)** *We propose to use a reduced product between the interval domain [12] and the affine equality domain [27]. This way, our abstract domain expresses constraints either of the form  $a \leq v \leq b$ , or of the form  $\sum a_k.v_k = b$ . Interval constraints (of the form  $a \leq v \leq b$  where  $a, b \in \mathbb{N}$  and  $v \in \mathcal{K}$ ) express properties of interest. Affine equalities (of the form  $\sum a_k.v_k = b$  where  $a_1, \dots, a_n, b \in \mathbb{Q}$ , and  $v_1, \dots, v_n \in \mathcal{K}$  express more complex properties, such as mutual exclusion. This allows for more precise calculations in the interval domain. Moreover, affine equalities capture relations when some threads are created and some others are consumed. To get a good precision, we need to avoid undetermined forms (when two unbounded values are subtracted) as much as possible. So, we use the approximate reduced product given in [22, Chap. 9], in which each primitive can be computed in  $\mathcal{O}(\text{Card}(\mathcal{K})^3)$  operations. Thus, we get a polynomial analysis.*

*Other domains could have been considered. The polyhedron domain [17] is too expensive. The octagon domain [30, 31] cannot express the affine invariants that are required when dealing with semaphores that both involve more than two agents and several tokens. Abstract multi-sets [32, 33] are exponential in time.*  $\square$

**Example 9.2 (the shared memory (cont.))** *We apply our content analysis on the example of the shared-memory (e.g. see example 4.1) with the reduced product of intervals and affine equalities (e.g. see example 9.1). We denote by  $\text{CU}$  the result of our analysis. The constraint system  $\text{CU}(\text{cell})$  describes the usage of channels opened by the instances of the restriction  $\nu \text{ cell}$ . Our goal is to prove that the system  $\text{CU}(\text{cell})$  entails both the affine equality constraint  $x_2 + x_6 + x_{10} = y_{1,13}$  and the interval constraint  $0 \leq y_{1,13} \leq 1$ . This means*

that either the channel has not been opened yet (i.e.  $y_{1,13} = 0$ ), or the channel has been opened (i.e.  $y_{1,13} = 1$ ) and there is exactly one output over it at the program point **2**, **6**, or **10** (since  $x_2 + x_6 + x_{10} = 1$ ).  $\square$

Now, we simulate the non-standard semantics in the abstract.

## 9.2 Initial state

At the beginning of the concrete computation, each variable  $x$  is bound to the name  $(x, \varepsilon)$ . Besides, the configuration contains one thread at each program point the label of which is in the set  $\beta(\mathcal{S})$ . Thus, a thread at program point  $l$  is in the computation unit  $[b \mapsto (\text{getvar}(l)(b), \varepsilon)]$ . So, at the beginning of the computation, a computation unit  $u$  is either empty, or it contains a thread at each program point  $l \in \beta(\mathcal{S})$  such that  $\alpha_{\text{UNIT}}(u) = \text{getvar}(l)$ . In the abstract, we introduce a primitive  $\chi_{\mathcal{N}(\mathcal{K})} \in \wp(\mathcal{K}) \rightarrow \mathcal{N}(\mathcal{K})$ . For any set  $A \in \wp(\mathcal{K})$ , we denote by  $\chi(A)$  the characteristic function of  $A$  which maps any variable  $v \in \mathcal{K}$  to 1 whenever  $v \in A$ , and to 0 otherwise. We require that  $\chi(A) \in \gamma_{\text{CON}}(\chi_{\mathcal{N}(\mathcal{K})}(A))$ .

**Example 9.3 (interval and affine constraints (cont.))** *In our abstract domain, the primitive  $\chi_{\mathcal{N}(\mathcal{K})}$  maps any set  $A \subseteq \mathcal{K}$  of variables, to the set of constraints  $\{v = 1 \mid v \in \mathcal{K}\} \cup \{v = 0 \mid v \notin \mathcal{K}\}$ .  $\square$*

The abstract state  $\mathcal{I}_{\text{CON}}^\sharp$  is defined in Fig. 3(a) as the function mapping any abstract computation unit  $a \in \text{UNIT}^\sharp$  to the element  $\sqcup_{\mathcal{N}(\mathcal{K})} \{\chi_{\mathcal{N}(\mathcal{K})}(\{x_l \mid l \in \beta(\mathcal{S}), \text{getvar}(l) = a\}); \chi_{\mathcal{N}(\mathcal{K})}(\emptyset)\}$ .

**Example 9.4 (the shared memory (cont.))** *In the shared memory example (e.g. see Ex. 4.1), the abstract element  $\mathcal{I}_{\text{CON}}^\sharp$  is equal to:*

$$\begin{cases} \text{alloc} & \mapsto \{0 \leq x_1 \leq 1\} \cup \{v = 0 \mid \forall v \in \mathcal{K} \setminus \{x_1\}\}, \\ \text{rec}_{12} & \mapsto \left\{ \begin{array}{l} 0 \leq x_{12} \leq 1 \\ x_{12} = x_{12'} \end{array} \right\} \cup \{v = 0 \mid \forall v \in \mathcal{K} \setminus \{x_{12}; x_{12'}\}\}, \\ - & \mapsto \{v = 0, \forall v \in \mathcal{K}\}; \end{cases}$$

since we have:  $\beta(\mathcal{S}) = \{1, 12, 12'\}$ ,  $\text{getvar}(1) = \text{alloc}$ , and  $\text{getvar}(12) = \text{getvar}(12') = \text{rec}_{12}$ .  $\square$

## 9.3 Transition step

We consider an abstract element  $\text{CU} \in \mathcal{C}_{\text{CON}}^\sharp$ , two program point labels  $l_?$  and  $l_l$ , and a transition sub-case  $(\sim, A) \in \text{CONTEXT}(l_?, l_l)$ . We simulate in the abstract any computation step  $\tau$  that matches  $C \xrightarrow{\lambda} C'$ , where  $\lambda = ((l_?, l_l), (\sim, A))$  (e.g. see Fig. 3(b)).

**Example 9.5 (the shared memory (cont.))** *As a running example, we simulate an interaction between a thread at the program point **5** and a thread at the*



$$\mathcal{I}_{\text{CON}}^{\#} := [a \mapsto \sqcup_{\mathcal{N}(\mathcal{K})} \{ \chi_{\mathcal{N}(\mathcal{K})}(\{x_l \mid l \in \beta(\mathcal{S}) \mid \mathbf{getvar}(l) = a\}), \chi_{\mathcal{N}(\mathcal{K})}(\emptyset) \}].$$

(a) Initial configuration abstraction.

Let  $l_?$  and  $l_!$  be two program point labels in  $\mathcal{L}$ , such that  $\mathbf{type}(l_?) \in \{\mathit{input}, \mathit{fetch}\}$ ,  $\mathbf{type}(l_!) = \mathit{output}$ , and such that the length of the lists  $\mathbf{arg}(l_?)$  and  $\mathbf{arg}(l_!)$  is the same. Let  $(\sim, A) \in \text{CONTEXT}(l_?, l_!)$  be a partition case and  $\text{CU} \in \mathcal{C}_{\text{CON}}^{\#}$  be an abstract element. We define  $\text{POST}_{\text{CON}}(\text{CU}, ((l_?, l_!), (\sim, A)))$  by  $\perp_{\text{CON}}$ , whenever there exists  $\diamond \in \{l_?, l_!\}$  such that  $\text{SYNC}_{\text{CON}}([l_{\diamond}, \diamond]_{\sim}) \cap \{(l_?, ?); (l_!, !)\}(\text{CU}(A([l_{\diamond}, \diamond]_{\sim}))) = \perp_{\mathcal{N}(\mathcal{K})}$ ; otherwise, we define it by  $[a \mapsto \sqcup_{\mathcal{N}(\mathcal{K})} \{\text{CU}(a)\} \cup \{\text{content}_1(P) \mid P \in (\mathcal{T}(l_?, l_!))_{\sim}, A(P) = a\}]$ , where, for any  $P \in (\mathcal{T}(l_?, l_!))_{\sim}$ :

- $\text{old}(P) := \chi_{\mathcal{N}(\mathcal{K})}(\emptyset)$ ,  
whenever  $\{\mathbf{getvar}(l)(b) \in \mathbf{I}(l) \setminus \mathbf{fv}(\mathbf{cont}(l_{\diamond})) \mid (l, \diamond) \in P, b \in B\} \neq \emptyset$ , or
  - $\text{SYNC}_{\text{CON}}(P \cap \{(l_?, ?); (l_!, !)\})(\text{CU}(A(P)))$ , otherwise;
  - $\text{consumed}_?(P) := \begin{cases} \{l_?\} & \text{whenever } \mathbf{type}(l_?) = \mathit{input} \text{ and } (l_?, ?) \in P, \\ \emptyset & \text{otherwise;} \end{cases}$
  - $\text{consumed}_!(P) := \begin{cases} \{l_!\} & \text{whenever } (l_!, !) \in P, \\ \emptyset & \text{otherwise;} \end{cases}$
  - $\text{created}_?(P) := \{l \mid l \neq l_?, (l, ?) \in P\}$  and  $\text{created}_!(P) := \{l \mid l \neq l_!, (l, !) \in P\}$ ;
  - $\text{content}_0(P) := \text{old}(P) -^{\#} (\chi_{\mathcal{N}(\mathcal{K})}(\text{consumed}_? \cup \text{consumed}_!)) +^{\#} (\chi_{\mathcal{N}(\mathcal{K})}(\text{created}_? \cup \text{created}_!))$ ;
  - $\text{content}_1(P) := \text{update\_trans}(l_?, l_!)(\text{content}_0(P))$ .
- (b) Abstract POST operator.

Figure 3: Contents analysis.

program point **10**. We start from an abstract element  $\text{CU}$  such that the system  $\text{CU}([b \mapsto \text{cell}])$  is made of both the constraints  $x_2 + x_6 + x_{10} = y_{1,13}$  and  $0 \leq y_{1,13} \leq 1$ .

We set  $l_? = 5$  and  $l_! = 10$ . Thanks to the control flow analysis, we only take into account the transitions where  $(5, ?) \sim (6, ?)$ ,  $(5, ?) \sim (10, !)$ ,  $(5, ?) \not\sim (7, ?)$ ,  $A([(5, ?)]_\sim)(b) = \text{alloc}$ , and  $A([(7, ?)]_\sim)(b) = \text{return}$ . Indeed, results coming from the other cases are ignored thanks to the coalesced product (e.g. see 7.3).  $\square$

### 9.3.1 Is the step possible ?

First, we check whether the computation step is possible, or not. Whenever we have  $(l_?, ?) \sim (l_!, !)$ , there must be a computation unit  $u$  in  $C$  such that both  $\alpha_{\text{UNIT}}(u) = A([(l_?, ?)]_\sim)$  and  $u$  contains at least one thread at the program point  $l_?$  and one thread at the program point  $l_!$ ; whenever we have  $(l_?, ?) \not\sim (l_!, !)$ , there must be two computation units  $u_?$  and  $u_!$  such that: for any  $\diamond \in \{?, !\}$ ,  $\alpha_{\text{UNIT}}(u_\diamond) = A([(l_\diamond, \diamond)]_\sim)$  and  $u_\diamond$  contains at least a thread at the program point  $l_\diamond$ . To check these properties, we require an abstract primitive  $\text{SYNC}_{\text{CON}} \in \wp(\mathcal{K}) \rightarrow \mathcal{N}(\mathcal{K}) \rightarrow \mathcal{N}(\mathcal{K})$  to check whether some variables may simultaneously take a non-zero value. For any set  $I$  of variables and any abstract element  $a \in \mathcal{N}(\mathcal{K})$ , the set  $\{f \in \gamma_{\mathcal{N}(\mathcal{K})}(a) \mid \forall v \in I, f(v) \geq 1\}$  should be included in the concretization  $\gamma_{\mathcal{N}(\mathcal{K})}(\text{SYNC}_{\text{CON}}(I)(a))$ . If there exists  $\diamond \in \{?, !\}$  such that  $\text{SYNC}_{\text{CON}}([(l_\diamond, \diamond)]_\sim) \cap \{(l_?, ?); (l_!, !)\}(\text{CU}(A([(l_\diamond, \diamond)]_\sim)))$  is equal to the bottom element  $\perp_{\mathcal{N}(\mathcal{K})}$ , the computation step is not possible, so we define  $\text{POST}_{\text{CON}}(\text{CU}, ((l_?, l_!), (\sim, A)))$  as  $\perp_{\text{CON}}$ . Otherwise, we update the abstraction of any computation unit involved in the computation step.

**Example 9.6 (the shared memory (cont.))** We know that i)  $(5, ?) \sim (10, !)$  and ii)  $A([(5, ?)]_\sim) = [b \mapsto \text{cell}]$ . We compute  $t$  that is defined by the expression  $\text{SYNC}_{\text{CON}}(\{(5, ?); (10, !)\})(\text{CU}([b \mapsto \text{cell}]))$ . The system  $t$  is equivalent to the system:

$$\left\{ x_5 \geq 1, x_{10} \geq 1, x_2 + x_6 + x_{10} = y_{1,13}, 0 \leq y_{1,13} \leq 1. \right.$$

By reduction, we obtain that  $t$  is equivalent to the system:

$$\left\{ x_5 \geq 1, y_{1,13} = x_{10} = 1, x_6 = x_2 = 0. \right.$$

This means that the interaction is only enabled when the cell has already been created ( $y_{1,13} = 1$ ) and when both interacting threads are in the computation unit ( $x_5 \geq 1$  and  $x_{10} = 1$ ). In this case, there is no thread at either the program point **2**, or at the program point **6** ( $x_6 = x_2 = 0$ ).

### 9.3.2 Abstracting the former contents of partition classes

Let us consider a class  $P \in (\mathcal{T}(l_?, l_!))_\sim$ . The class  $P$  denotes a computation unit  $u$  that is transformed during the computation step. We first compute an abstraction  $\text{old}(P)$  of the contents of  $u$  before the computation step.

In the case where there exists a pair  $(l, \diamond) \in P$  and a key  $b \in B$  such that  $\text{getvar}(l)(b) \in \mathbf{I}(l) \setminus \text{fv}(\text{cont}(l_\diamond))$ , the computation unit maps a key to a fresh name, so we can deduce that the computation unit  $u$  has been created during the transition step. In such a case, we define  $\text{old}(P)$  as  $\chi_{\mathcal{N}(\mathcal{K})}(\emptyset)$ . Otherwise, we take into account the abstraction of the computation unit and the threads that are required to enable the computation step: we define  $\text{old}(P)$  as  $\text{SYNC}_{\text{CON}}(P \cap \{(l_?, ?); (l_!, !)\})(\text{CU}(A(P)))$ .

**Example 9.7 (the shared memory (cont.))** *First, we compute the contents of partition class  $[(5, ?)]_\sim$  before the computation step. The element  $\text{old}([(5, ?)]_\sim)$  is equal to  $\text{SYNC}_{\text{CON}}([(5, ?)]_\sim \cap \{(5, ?); (10, !)\})(\text{CU}(A([(5, ?)]_\sim)))$ , so the system  $\text{old}([(5, ?)]_\sim)$  contains the constraints  $x_5 \geq 1$ ,  $x_{10} \geq 1$ ,  $x_2 + x_6 + x_{10} = y_{1,13}$ , and  $0 \leq y_{1,13} \leq 1$ . By reduction, we obtain that the system  $\text{old}([(5, ?)]_\sim)$  is given by the constraints  $x_5 \geq 1$ ,  $y_{1,13} = x_{10} = 1$ ,  $x_6 = x_2 = 0$ . This means that the interaction is only enable when the cell has already been created ( $y_{1,13} = 1$ ) and if the interacting threads are in the computation unit ( $x_5 \geq 1$  and  $x_{10} = 1$ ). In such a case, there is no thread at the program point **2** or at the program point **6** ( $x_6 = x_2 = 0$ ).  $\square$*

**Example 9.8 (the shared memory (cont.))** *We now consider a case when a computation unit is necessarily empty. We simulate an interaction between a thread at the program point **1** and a thread at the program point **13**. This way, we set  $l_? = 1$  and  $l_! = 13$ . Thanks to the control flow analysis, we only take into account the transitions where  $(l_?, ?) \sim (l_!, ?)$  and  $A([(l_?, ?)]_\sim)(b) = \text{alloc}$ . The interaction launches a thread at the program point **2**. But, we have  $\text{getvar}(2)(b) = \text{cell}$ ,  $\mathbf{I}(2) = \{\text{cell}; \text{null}\}$ , and  $\text{fv}(\text{cont}(1)) = \{\text{null}; \text{add}\}$ . So  $\text{cell} \in \mathbf{I}(2) \setminus \text{fv}(\text{cont}(1))$ . Thus we can conclude that  $\text{old}([(2, ?)]_\sim)$  is equal to  $\chi_{\mathcal{N}(\mathcal{K})}(\emptyset)$ . This way, the thread is launched in an empty computation unit.  $\square$*

### 9.3.3 Abstracting the evolution of partition classes

Then, we compute the set of labels of the threads that are created and consumed in the computation unit  $u$ . The input thread is consumed in  $u$  only if it is not a resource and if it was in the computation unit  $u$ : so we define  $\text{consumed}_?(P)$  as  $\{l_?\}$  if both  $\text{type}(l_?) = \text{input}$  and  $(l_?, ?) \in P$ , and as  $\emptyset$  otherwise. The output thread is always consumed (we only check whether it is in  $u$ , or not): so we define  $\text{consumed}_!(P) := \{l_!\}$  if  $(l_!, !) \in P$ , and  $\text{consumed}_!(P) := \emptyset$  otherwise. The threads that are created during the computation step are dealt with the same way: we define  $\text{created}_\diamond(P) := \{l \mid l \neq l_\diamond, (l, \diamond) \in P\}$ , for any  $\diamond \in \{?; !\}$ .

**Example 9.9 (the shared memory (cont.))** *In our running example, the set  $\text{consumed}_?([(5, ?)]_\sim)$  is equal to  $\{5\}$ , the set  $\text{consumed}_!([(5, ?)]_\sim)$  is equal to  $\{10\}$ . Since the constraints  $(5, ?) \sim (6, ?)$  and  $(5, ?) \not\sim (7, ?)$  are satisfied, we can deduce that the set  $\text{created}_?([(5, ?)]_\sim)$  is equal to  $\{6\}$ . Last, the set  $\text{created}_!([(5, ?)]_\sim)$  is empty.  $\square$*

The abstraction  $\text{content}_0(P)$  of the contents of the computation unit  $u$  after the computation step can then be defined as  $\text{old}(P) -^\# (\chi_{\mathcal{N}(\mathcal{K})}(\text{consumed}_? \cup \text{consumed}_!)) +^\# (\chi_{\mathcal{N}(\mathcal{K})}(\text{created}_? \cup \text{created}_!))$ , where  $+^\#$  and  $-^\#$  are sound counterparts to the point wise addition and to the point wise subtraction. More precisely, for any  $a, b \in \mathcal{N}(\mathcal{K})$  and  $\circ \in \{+, -\}$ , we have:  $a \circ^\# b \in \mathcal{N}(\mathcal{K})$ , and the concretization  $\gamma_{\mathcal{N}(\mathcal{K})}(a \circ^\# b)$  contains at least all functions  $[v \mapsto f(v) \circ g(v)]$  such that:  $f \in \gamma_{\mathcal{N}(\mathcal{K})}(a)$ ,  $g \in \gamma_{\mathcal{N}(\mathcal{K})}(b)$ , and for any  $v \in \mathcal{K}$ ,  $f(v) \circ g(v) \geq 0$ .

**Example 9.10 (interval and affine constraints (cont.))** *The primitives  $+^\#$  and  $-^\#$  are both computed pair-wise over the system of affine constraints and over the system of interval constraints. More details can be found in [22, Chap. 9, Sect. 9.3.1].  $\square$*

The last step consists in updating the local history of computation units. We introduce a primitive  $\text{update\_trans} \in \mathcal{L}^2 \rightarrow \mathcal{N}(\mathcal{K}) \rightarrow \mathcal{N}(\mathcal{K})$ . The function  $\text{update\_trans}(\lambda)$  increments, in the abstract, the value of variable  $y_\lambda$  and sets the value of variable  $z_\lambda$  to 1. So, for any function  $f \in \gamma_{\mathcal{N}(\mathcal{K})}(a)$ , the function  $g$  that maps  $y_\lambda$  to  $f(y_\lambda) + 1$ ,  $z_\lambda$  to 1, and any other variable  $v$  to  $f(v)$  should be an element of the concretization  $\gamma_{\mathcal{N}(\mathcal{K})}(\text{update\_trans}(\lambda)(a))$ . Thus, we define  $\text{content}_1(P)$  as  $\text{update\_trans}(l_?, l_!)(\text{content}_0(P))$ .

**Example 9.11 (interval and affine constraints (cont.))** *We can define the primitive  $\text{update\_trans}(\lambda)$  by using the usual transfer functions for assignments in interval constraints (e.g. see [12]) and in affine equalities (e.g. see [27]).  $\square$*

**Example 9.12 (the shared memory (cont.))** *In our running example, the system  $\text{content}_0([(5, ?)]_\sim)$  is given by the constraints  $x_5 \geq 0$ ,  $y_{1,13} = 1$ ,  $x_{10} = 0$ ,  $x_6 = 1$ ,  $x_2 = 0$ . Then,  $\text{content}_1([(5, ?)]_\sim)$  is given by the constraints  $x_5 \geq 0$ ,  $y_{1,13} = 1$ ,  $x_{10} = 0$ ,  $x_6 = 1$ ,  $x_2 = 0$ ,  $y_{5,10} \geq 1$ , and  $z_{5,10} = 1$ .  $\square$*

### 9.3.4 Updating abstract elements

We are left to update the abstraction of the computation units whose abstract computation unit is  $A(P)$ . We define, for any  $a \in \text{UNIT}^\#$ ,  $\text{POST}_{\text{CON}}(\text{CU}, ((l_?, l_!), (\sim, A))) (a)$  as  $\sqcup_{\mathcal{N}(\mathcal{K})} \{\text{CU}(a)\} \cup \{\text{content}_1(P) \mid P \in (\mathcal{T}(l_?, l_!))_\sim, A(P) = a\}$ .

**Example 9.13 (the shared memory (cont.))** *We recall the fact that the system  $\text{CU}([b \mapsto \text{cell}])$  entails the affine constraints  $x_2 + x_6 + x_{10} = y_{1,13}$  and the interval constraint  $0 \leq y_{1,13} \leq 1$ . The class  $P = [(5, ?)]_\sim$  is the only one such that  $A(P) = [b \mapsto \text{cell}]$ . Moreover, the affine constraints  $x_2 + x_6 + x_{10} = y_{1,13}$  and the interval constraint  $0 \leq y_{1,13} \leq 1$  are also entailed by the system  $\text{content}_1(P)$ . The analysis discovers that these constraints are invariant.  $\square$*

## 9.4 Soundness

Thm. 9.14 states the soundness of our content analysis.

**Theorem 9.14**  $(\mathcal{C}_{con}^\sharp, \sqcup_{con}, \perp_{con}, \gamma_{con}, \mathcal{I}_{con}^\sharp, \text{POST}_{con}, \nabla_{con})$  is an abstraction.

## 10 Conclusion

We have proposed a generic framework for statically inferring properties of mobile systems. This framework is based on thread partitioning: we gather the threads of a mobile system into several classes. The criterion of thread partitioning is left as a parameter. We use the product of an analysis of the dynamic linkage between the threads of a system and an analysis of the number of threads inside each partition class. As a result, we get a polynomial-time (with respect to the length of the initial state) analysis, which succeeds in proving the absence of race conditions in a shared memory written in the  $\pi$ -calculus. In [22, Chap:10], we propose a version of this framework for the *ambient*-calculus (see. Sect. 10.2), and a model independent version (see. Sect. 10.3). We succeed in proving authentication properties in a version [38] of the Woo and Lam one-way public-key authentication protocol that is written in the *spi*-calculus [1]. For that purpose, we partition the threads according to the identities of the principals that have initiated the session.

Thread partitioning may also be used in *reconfigurable systems* to prove that the system may not switch to a new version until all components have been installed. For that purpose, we may partition threads according to the version identifier. As future works, we are also interested in using thread partitioning to refine the type checking of authorization policies [23].

## References

- [1] M. Abadi and A.D. Gordon. A calculus for cryptographic protocols: The spi calculus. *Information and Computation*, 148(1), 1999.
- [2] Gérard Berry and Gérard Boudol. The chemical abstract machine. *Theor. Comput. Sci.*, 96(1):217–248, 1992.
- [3] Bruno Blanchet. From Secrecy to Authenticity in Security Protocols. In Manuel Hermenegildo and Germán Puebla, editors, *9th International Static Analysis Symposium, SAS'02*, volume 2477 of *Lecture Notes on Computer Science*, pages 342–359, Madrid, Spain, September 2002. Springer Verlag.
- [4] Chiara Bodei, Pierpaolo Degano, Flemming Nielson, and Hanne Riis Nielson. Control flow analysis for the pi-calculus. In *Proceedings of the 9th International Conference on Concurrency Theory, CONCUR '98*, pages 84–98, London, UK, 1998. Springer-Verlag.
- [5] Chiara Bodei, Pierpaolo Degano, Flemming Nielson, and Hanne Riis Nielson. Static analysis for the pi-calculus with applications to security. *Inf. Comput.*, 168(1):68–92, 2001.

- [6] François Bourdoncle. Abstract interpretation by dynamic partitioning. *J. Funct. Program.*, 2(4):407–423, 1992.
- [7] Luca Cardelli, Giorgio Ghelli, and Andrew D. Gordon. Ambient groups and mobility types. In *Proceedings of the International Conference IFIP on Theoretical Computer Science, Exploring New Frontiers of Theoretical Informatics, TCS '00*., pages 333–347, London, UK, 2000. Springer-Verlag.
- [8] Luca Cardelli, Giorgio Ghelli, and Andrew D. Gordon. Secrecy and group creation. *Information and Computation*, 196(2):127–155, 2005.
- [9] Luca Cardelli and Andrew D. Gordon. Mobile ambients. *Theoretical Computer Science*, 240(1), 1998.
- [10] Sagar Chaki, Sriram K. Rajamani, and Jakob Rehof. Types as models: Model checking message-passing programs. In *Proceedings of the 29th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '02*, 2002.
- [11] P. Cousot. *Méthodes itératives de construction et d'approximation de points fixes d'opérateurs monotones sur un treillis, analyse sémantique des programmes*. PhD thesis, Université Scientifique et Médicale de Grenoble, 1978.
- [12] Patrick Cousot and Radhia Cousot. Static determination of dynamic properties of programs. In *Proceedings of the Second International Symposium on Programming, POPL '76*, pages 106–130. Dunod, Paris, France, 1976.
- [13] Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the Fourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '77*, pages 238–252, Los Angeles, California, 1977. ACM Press, New York, NY.
- [14] Patrick Cousot and Radhia Cousot. Systematic design of program analysis frameworks. In *Conference Record of the Sixth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '79*, pages 269–282, San Antonio, Texas, 1979. ACM Press, New York, NY.
- [15] Patrick Cousot and Radhia Cousot. Abstract interpretation frameworks. *Journal of Logic and Computation*, 2(4):511–547, August 1992.
- [16] Patrick Cousot and Radhia Cousot. Comparing the Galois connection and widening/narrowing approaches to abstract interpretation, invited paper. In M. Bruynooghe and M. Wirsing, editors, *Proceedings of the International Workshop Programming Language Implementation and Logic Programming, PLILP '92*, Leuven, Belgium, 13–17 August 1992, Lecture Notes in Computer Science 631, pages 269–295. Springer-Verlag, Berlin, Germany, 1992.

- [17] Patrick Cousot and Nicolas Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Proceedings of the 5th ACM SIGACT-SIGPLAN symposium on Principles of programming languages, POPL '78*, pages 84–96, New York, NY, USA, 1978. ACM.
- [18] Jérôme Feret. Confidentiality analysis of mobile systems. In *Proceedings of the 7th International Symposium on Static Analysis, SAS '00*, pages 135–154, London, UK, 2000. Springer-Verlag.
- [19] Jérôme Feret. Occurrence counting analysis for the pi-calculus. *Electr. Notes Theor. Comput. Sci.*, 39(2), 2001.
- [20] Jérôme Feret. Dependency analysis of mobile systems. In *Proceedings of the 11th European Symposium on Programming Languages and Systems, ESOP '02*, pages 314–330, London, UK, 2002. Springer-Verlag.
- [21] Jérôme Feret. Abstract interpretation of mobile systems. *J. Log. Algebr. Program.*, 63(1):59–130, 2005.
- [22] Jérôme Feret. *Analysis of mobile systems by abstract interpretation*. PhD thesis, École Polytechnique, 2005.
- [23] Cédric Fournet, Andrew D. Gordon, and Sergio Maffei. A type discipline for authorization policies. *ACM Trans. Program. Lang. Syst.*, 29(5):25, 2007.
- [24] Roberta Gori and Francesca Levi. A new occurrence counting analysis for bioambients. In Kwangkeun Yi, editor, *Proceedings of Programming Languages and Systems, Third Asian Symposium, APLAS '05*, volume 3780 of *Lecture Notes in Computer Science*, pages 381–400. Springer, 2005.
- [25] Matthew Hennessy and James Riely. Resource access control in systems of mobile agents. In Uwe Nestmann and Benjamin C. Pierce, editors, *High-Level Concurrent Languages, HLCL '98*, volume 16.3, pages 3–17. Elsevier Science Publishers, 1998.
- [26] Atsushi Igarashi and Naoki Kobayashi. A generic type system for the Pi-calculus. *ACM SIGPLAN Notices*, 36(3):128–141, 2001.
- [27] Michael Karr. Affine relationships among variables of a program. *Acta Inf.*, 6:133–151, 1976.
- [28] Laurent Mauborgne and Xavier Rival. Trace partitioning in abstract interpretation based static analyzers. In M. Sagiv, editor, *European Symposium on Programming, ESOP '05*, volume 3444 of *Lecture Notes in Computer Science*, pages 5–20. Springer-Verlag, 2005.
- [29] Robin Milner. The polyadic pi-calculus: a tutorial. In F. L. Bauer, W. Brauer, and H. Schwichtenberg, editors, *Logic and Algebra of Specification*, pages 203–246. Springer-Verlag, 1993.

- [30] Antoine Miné. *Weakly Relational Numerical Abstract Domains*. PhD thesis, École Polytechnique, 2004.
- [31] Antoine Miné. The octagon abstract domain. *Higher Order Symbol. Comput.*, 19(1):31–100, 2006.
- [32] Flemming Nielson, René Rydhof Hansen, and Hanne Riis Nielson. Abstract interpretation of mobile ambients. *Science Computer Programming*, 47(2-3):145–175, 2003.
- [33] Hanne Riis Nielson and Flemming Nielson. Shape analysis for mobile ambients. In *Proceedings of the 21th International Symposium on Programming, POPL '00*, pages 142–154, 2000.
- [34] Sriram K. Rajamani and Jakob Rehof. A behavioral module system for the pi-calculus. In *Proceedings of the 8th International Symposium on Static Analysis, SAS '01*, pages 375–394, London, UK, 2001. Springer-Verlag.
- [35] A. Regev, E.M. Panima, W. Silverman, L. Cardelli, and E. Shapiro. Bioambients: An abstraction for biological compartments. *Theoretical Computer Science*, 325(1), 2004.
- [36] James Riely and Matthew Hennessy. A typed language for distributed mobile processes. In *Conference Record of POPL 98: The 25TH ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Diego, California*, pages 378–390, New York, NY, 1998.
- [37] David N. Turner. *The Polymorphic  $\pi$ -Calculus: Theory and Implementation*. PhD thesis, Edinburgh University, 1995.
- [38] Thomas Y. C. Woo and Simon S. Lam. Authentication for distributed systems. *Computer*, 25(1):39–52, 1992.